PATENT ABSTRACTS OF JAPAN

(11)Publication number : 2000-057001
(43)Date of publication of application : 25.02.2000

(51)Int.Cl. G06F 11/22
G01R 31/28
G06F 11/26
G06F 17/50

(21)Application number : 11-147226
(22)Date of filing : 26.05.1999

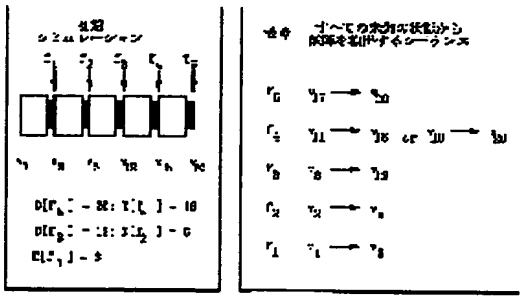
(71)Applicant : NEC CORP
(72)Inventor : BOMMU SURENDRA K
CHAKRADHAR SRIMAT
DORESWAMY KIRAN

(30)Priority

Priority number : 98 86758 Priority date : 27.05.1998 Priority country : US
98 112945 10.07.1998 US

(54) VECTOR RESTORING METHOD USING ACCELERATED VERIFICATION AND IMPROVEMENT

(57)Abstract:
PROBLEM TO BE SOLVED: To shorten the executing time of vector restoring by recognizing the shortest substring that detects a failure object among those substrings existing between the 1st and 2nd substrings in a restoration phase.
SOLUTION: A satisfactory substring of a vector is quickly detected in a verification phase. A substring that can detect all failure objects is assured to include even short substrings of the vector. In an improvement phase, the unnecessary parts of a verified segment are deleted and the shortest substring that detects all failure objects is found out. In a restoration process, it's known that a failure can be detected via the substrings which are not overlapping with each other. That is, the failures f3 and f5 can be restored in the shown examples. Thus, a 1st substring of the vector which detects a failure object and a 2nd substring which detects no failure object are identified in the verification phase. Then the shortest substring that detects a failure object is recognized in the restoration phase among those substrings existing between both 1st and 2nd substrings.



LEGAL STATUS
[Date of request for examination] 21.04.2000

Searching PAJ

[Date of sending the examiner's decision of rejection]	
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]	
[Date of final disposal for application]	
[Patent number]	3275309
[Date of registration]	08.02.2002
[Number of appeal against examiner's decision of rejection]	
[Date of requesting appeal against examiner's decision of rejection]	
[Date of extinction of right]	

(19) BUNDESREPUBLIK DEUTSCHLAND

DEUTSCHES PATENT- UND MARKENAMT

(12) **Offenlegungsschrift**

(10) **DE 199 24 242 A 1**

(51) Int. Cl.[6]:
G 06 F 11/00
G 06 F 17/00
G 06 F 17/50
G 01 R 31/3183

(21) Aktenzeichen: 199 24 242.9
(22) Anmeldetag: 27. 5. 99
(43) Offenlegungstag: 30. 12. 99

(30) Unionspriorität:
60/086,758 27. 05. 98 US
09/112,945 10. 07. 98 US

(71) Anmelder:
NEC Corp., Tokio/Tokyo, JP

(74) Vertreter:
Pätzold, H., Dipl.-Ing. Dr.-Ing., Pat.-Anw., 82166 Gräfelfing

(72) Erfinder:
Bommu, Surenda K., Princeton, N.J., US;
Chakradhar, Srimat T., Princeton, N.J., US;
Doreswamy, Kiran B., Princeton, N.J., US

Die folgenden Angaben sind den vom Anmelder eingereichten Unterlagen entnommen

(54) Vektorrestauration mit beschleunigter Validation und Verfeinerung

(57) Ein Zweiphasenvektorrestaurationsverfahren, das eine minimale Teilsequenz aus einer Sequenz extrahiert, die eine ausgewählte Menge von Fehlern detektiert. Das Vektorrestaurationsverfahren wird bei statischen Kompaktierungsverfahren von Testsequenzen und bei Fehlerdiagnoseverfahren verwendet. Eine beschleunigte Version des Zweiphasenvektorrestaurationsverfahrens bietet eine weitere Verbesserung. Die vorliegende Erfindung ist dem Stand der Technik deutlich überlegen; 1. eine Sequenz einer Länge n kann mit nur $0 (n \log_2 n)$-Simulationen simuliert werden, während herkömmliche Verfahren Simulationen von $0(n^2)$-Vektoren benötigen; 2. ein zweistufiges Restaurationsverfahren wird verwendet, so daß die Vektorrestauration auch auf große Designs anwendbar ist, und 3. das Restaurationsverfahren für mehrere Fehler wird überlappt und eine deutliche Beschleunigung der Vektorrestauration erzielt. Das erfindungsgemäße Vektorrestaurationsverfahren kann in ein statisches Testsequenzkompaktierungssystem integriert werden.

DE 199 24 242 A 1

Beschreibung

BESCHREIBUNG DER ERFINDUNG

Die vorliegende Erfindung beansprucht die Priorität der parallelen anhängigen US-Provisional-Patentanmeldung Ser. Nr. 60/086,758 vom 27. Mai 1998.

BEREICH DER ERFINDUNG

Die vorliegende Erfindung betrifft ein Restaurationsverfahren von Testvektoren, die zur Fehlerermittlung in großen industriellen Design-Systemen verwendet werden. Die vorliegende Erfindung betrifft insbesondere ein Restaurationsverfahren einer minimalen Teilsequenz von Vektoren, die eine gegebene Fehlermenge detektieren. Ein Ausführungsbeispiel der vorliegenden Erfindung betrifft ein Verfahren einer zweistufigen Vektorrestauration und ein Verfahren für eine beschleunigte zweistufige Vektorrestauration, und verbessert die Laufzeiten von Fehlerdiagnosen für große Designs.

HINTERGRUND DER ERFINDUNG

Ein Vektor ist eine Menge von Eingaben in ein System. Eine Testmenge ist eine Menge von Vektoren, die Fehler in dem System erkennen. Eine Zielfehlerliste ist eine Liste einer Teilmenge von Fehlern, die von einer gegebenen Testmenge identifiziert werden. Mit einer gegebenen Testmenge und einer Menge einer Zielfehlerliste von der bekannt ist, daß sie von der Testmenge detektiert zu werden, identifizieren Vektorrestaurationsverfahren eine minimale Teilsequenz, die alle Fehler in der Zielfehlerliste erkennt. Siehe R. Guo, I. Pomeranz, und S. M. Reddy "Procedures for static compaction of test Seguences for synchronous Sequential circuits based on vector restoration", Technical Report 3.08.1997, Electrical and Computer Engineering Department, University of lowa, 1997.

Restaurationsverfahren werden in statischen TestSequenz-Kompaktierungsverfahren oder Fehlerdiagnoseverfahren verwendet Herkömmliche statische TestSequenz-Kompaktierungsverfahren werden in der nachfolgenden Literatur beschrieben: T. M. Niermann, R. K. Roy, J. H. Patel und J. A. Abraham "Test compaction for Sequential circuits", IEEE Trans. Computer-Aided Design, Vol. 11, Nr. 2, S. 260–267, Februar 1992; B. So "Time-efficient automatic test pattern generation system", Ph. D. Thesis, EE Dept., Univ. of Wisconsin at Madison, 1994; I. Pomeranz und S. M. Reddy "On static compaction of test Sequences for snychronous Sequential circuits", Proc. Design Automation Conf., S. 215–220, Juni 1996; M. S. Hsiao, E. M. Rudnick und J. H. Patel "Fast algorithms for static compaction of Sequential circuit test vectors", Proc. IEEE VLSI Test Symp., S. 188–195, April 1995; S. T. Chakradhar und M. S. Hsiao, "Partitioning and Reordering Techniques for Static Test Sequence Compaction of Sequential Circuits", Technical Report 1997, Computers & Communications Research Lab., NEC USA Inc.

Kompaktierungsverfahren auf der Grundlage von Vektorrestauration werden beschrieben in I. Pomeranz und S. M. Reddy "Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential circuits". Proceedings Int. Conf. on Computer Design, S. 360–365, 1997, University of lowa, August 1997; und A. Raghunathan und S. T. Chakradhar "Acceleration techniques for dynamic vector compaction", Proc. Int. Conf. Computer Aided Design, S. 310–317, August 1995.

Vektorrestauration, ein Überblick.

Eine Testmenge ist eine Sequenz von Vektoren $v_1, \ldots, v_n$ mit vorgegebener Reihenfolge. Die Testmenge detektiert Fehler $f_1, \ldots f_z$, die eine Fehlermenge F bilden. Wenn ein Fehler f von einem Vektor v_i der Testmenge erkannt wird, dann ist die Detektionszeit $D_{[f]}$ des Fehlers gleich i. Die Detektionszeiten können auf einfache Weise mittels einer vorgeschalteten Verarbeitungsphase ermittelt werden, die eine Fehlersimulation durchführt. Eine derartige Fehlersimulation wird herkömmlich mittels Fehlerabzweigung (fault dropping) durchgeführt.

Beispielsweise hat die in **Fig. 1** dargestellte Testmenge **20** Vektoren $v_1, \ldots, v_{20}$. Diese Testmenge erkennt fünf Fehler $f_1, \ldots f_5$. Der Fehler f_5 wird von dem Vektor v_{20} erkannt. Daher ist $D[f_5] = 20$. **Fig. 1** zeigt die Detektionszeiten von anderen Fehlern. Für einen gegebenen Fehler f mit $D[f] = t$ existiert eine Sequenz von Vektoren $v_i, \ldots, v_t$ $(1 \leq i \leq t)$, die den Fehler unter der Annahme eines unbekannten ursprünglichen Zustandes erkennt. Beispielsweise wird f_4 bei Vektor v_{16} erkannt, und es existiert eine Teilsequenz $v_1, \ldots, v_{16}$, die den Fehler erkennt. Es ist auch möglich, daß eine kürzere Sequenz von Vektoren $v_i, \ldots, v_t$ $(1 < i \leq t)$ den Fehler erkennt.

Ein herkömmliches lineares Vektorrestaurationsverfahren ermittelt die Teilsequenz, indem zunächst nur der Vektor v_t berücksichtigt wird. Wenn ein oder mehr Zielfehler nicht erkannt werden, dann wird die Teilsequenz v_{t-1}, v_t vorgeschlagen. Wenn diese Sequenz auch nicht alle Zielfehler erkennt, dann werden zusätzliche Vektoren $v_{t-2}, \ldots, v_1$ (in dieser Reihenfolge) berücksichtigt, bis alle Zielfehler erkannt sind. Ein derartiges Verfahren ist sehr komplex. Nachstehend wird die Komplexität dieses Verfahrens detaillierter beschrieben.

Fig. 1 zeigt ein Beispiel des linearen Vektorrestaurationsverfahrens mit Teilsequenzrestauration für Fehler f_4 und f_5, die Zielfehler sind. Da f_5 bei v_{20} erkannt wird, ist v_{20} die erste Teilsequenz, die von dem linearen Vektorrestaurationsverfahren berücksichtigt wird. Der Fehler f_5 wird von dieser ersten Teilsequenz nicht erkannt. Daher wird die Teilsequenz aktualisiert und geändert, so daß sie den Vektor v_{19} enthält. Die neue Teilsequenz ist v_{19}, v_{20}. Diese Teilsequenz erkennt aber den Fehler f_5 auch nicht. Daher werden zusätzliche Vektoren restauriert, bis die restaurierte Teilsequenz v_{17} enthält. Diese Teilsequenz erkennt den Fehler f_5, wie in **Fig. 1** dargestellt ist. Die Vektorrestauration für den Fehler f_5 ist damit abgeschlossen.

Der nächste Schritt ist die Überprüfung, ob f_4 ebenso von der restaurierten Vektormenge erkannt wird. Beispielsweise erkennt die Sequenz $v_{17}, \ldots, v_{20}$ ebenso f_4. Das Restaurationsverfahren für die Zielfehler f_4 und f_5 ist damit abgeschlossen.

Anwendung von Vektorrestauration

In einer US-Patentanmeldung von Bommu et al. ist ein statisches Kompaktierungsverfahren unter Verwendung von Vektorrestauration detailliert beschrieben.

Statische Kompaktierungsverfahren auf der Grundlage von Vektorrestauration verwenden das in **Fig. 2** skizzierte Verfahren. Bei einer gegebenen Testmenge, einer gegebenen Fehlermenge und einer gegebenen Detektionszeit für jeden Fehler stellen diese Verfahren eine kleinere Testmenge her, die wenigstens soviele Fehler erkennt, wie die ursprüngliche Testmenge. Die eingegebenen Daten enthalten eine Menge von Testvektoren, eine Liste mit zu detektierenden Fehlern, und die Detektionszeiten für die Fehler, wie in Block **2.01** von **Fig. 2** dargestellt ist. Zunächst wird in Block **2.02** eine Fehlermenge als Zielfehler ausgewählt. Diese Fehler können gleiche oder unterschiedliche Detektionszeiten haben.

Wenn die längste (latest) Detektionszeit eines Fehlers in der Zielliste t ist, dann findet das Restaurationsverfahren von Block **2.05** eine Teilsequenz von $v_i \ldots, v_t$, $(1 \leq i \leq t)$, die (1) alle Fehler in der Zielfehlerliste erkennt, und (2) falls die Teilmenge der restaurierten Teilmenge für frühere Zielfehler vorsteht, dann werden auch noch alle früheren Zielfehler detektiert. Die nächste Menge der Zielfehler wird aus den Fehlern ausgewählt die von den restaurierten Vektoren in Block **2.02** nicht erkannt werden. Dieses Verfahren wird in den Blöcken **2.03** und **2.04** wiederholt, bis alle Zielfehler erkannt sind.

Anhand der Nachteile der in **Fig. 1** dargestellten Kompaktierung einer Testmenge kann das Vektorrestaurationsverfahren erläutert werden. Die Vektorrestauration für Fehler f_4 und f_5 führt zu der restaurierten Sequenz $v_{17}, \ldots, v_{20}$. Fehler f_3 ist jedoch noch nicht erkannt. Daher wird als nächstes Zielfehler f_3 gewählt. Das Verfahren wird mit der Restauration für den Fehler f_3 fortgeführt. Die Detektionszeit für f_3 ist 12. Daher beginnt die Restauration von f_3 von dem Vektor v_{12} anstelle des Vektors v_{17}. Die erste für f_3 vorgeschlagene Sequenz ist $v_{12}, v_{17}, \ldots, v_{20}$. Das kommt daher, da es möglich ist, daß eine Sequenz, die mit v_{12} beginnt und mit irgendeinem Vektor der schon restaurierten Vektoren $v_{17}, \ldots, v_{20}$ endet, f_3 erkennen könnte. Daher werden bei der Restauration einer VektorSequenz für f_3 auch die schon restaurierten Vektoren $v_{17}, \ldots, v_{20}$ simuliert. Das Restaurationsverfahren wird fortgeführt, bis alle Fehler erkannt sind. Hier ist beispielsweise die kompaktierte Vektormenge $v_1, \ldots, v_{12}, v_{17}, \ldots, v_{20}$.

Die herkömmlichen linearen Vektorrestaurationsverfahren haben einen schwerwiegenden Nachteil, der ihre Verwendung einschränkt. Sie benötigen lange Laufzeiten bei ihrem Einsatz für große industrielle Designs.

ZUSAMMENFASSUNG DER ERFINDUNG

Es ist daher eine Aufgabe der vorliegenden Erfindung, ein verbessertes Vektorrestaurationsverfahren bereitzustellen, das erheblich geringere Laufzeiten benötigt, als herkömmliche Verfahren.

Es ist insbesondere eine Aufgabe der vorliegenden Erfindung, ein Restaurationsverfahren für eine minimale Teilsequenz von Testvektoren bereitzustellen, um ein System zu testen, das eine Fehlermenge hat, die von der Sequenz der Testvektoren erkennbar ist.

Die vorstehenden Aufgaben werden mit den Merkmalen der Ansprüche gelöst. Die vorstehenden Aufgaben werden insbesondere gelöst, indem ein Restaurationsverfahren einer Sequenz von Testvektoren zur Prüfung eines Systems bereitgestellt wird, wobei das System eine Fehlermenge hat, die von der Sequenz der Testvektoren detektierbar ist, und wobei eine Teilmenge der Fehlermenge Zielfehler genannt wird, und wobei das Verfahren eine Validationsphase und eine Restaurationsphase hat, und wobei in der Validationsphase eine erste Teilsequenz von Testvektoren erkannt wird, die die Zielfehler erkennen, und eine zweite Teilsequenz identifiziert wird, die die Zielfehler nicht erkennen, und wobei die Restaurationsphase die kürzeste Teilsequenz zwischen der ersten Teilsequenz und der zweiten Teilsequenz identifiziert, die die Zielfehler erkennt.

Ein anderer Aspekt der vorliegenden Erfindung betrifft ein Restaurationsverfahren einer Sequenz von Testvektoren mit den nachfolgenden Verfahrensschritten:

Zuordnung von Fehlern zu einer Fehlerliste;

Identifikation einer Detektionszeit für jeden der Fehler;

Initialisierung eines Restaurationssequenz-Listenendes (nil); Fehlerzuordnung aus der Fehlerliste mit höchsten Detektionszeiten zu einer Zielfehlerliste;

base wird auf das Minimum der höchsten Detektionszeiten gesetzt und auf eine Zeit, die einem ersten Vektor in der Restaurationssequenzliste entspricht;

Durchführung einer Validationsphase, wobei eine low-Teilsequenz der Testvektoren identifiziert wird, die alle Fehler der Zielfehlerliste erkennt, und eine high-Teilsequenz identifiziert wird, die keinen Fehler in der Zielfehlerliste erkennt;

Durchführung einer Verfeinerungsphase, wobei eine kürzeste Teilsequenz zwischen der low-Teilsequenz und der high-Teilsequenz identifiziert wird, wobei die kürzeste Teilsequenz alle Fehler in der Zielfehlerliste erkennt;

Entfernung der Fehler von der Fehlerliste, die auch in der Zielfehlerliste sind;

Aktualisierung der Restaurationssequenzliste mit der Vereinigung der Restaurationssequenzliste und der kürzesten identifizierten Teilsequenz in Schritt g; und

Wiederholung bis die Fehlerliste leer ist.

Eine vorteilhafte Ausführung umfaßt ein Verfahren, bei dem in der Validationsphase zusätzlich Vektoren kontinuierlich zu der Restaurationssequenz hinzugefügt werden, und eine Fehlersimulation durchgeführt wird, bis alle Fehler der Zielfehlerliste erkannt sind.

Eine weitere vorteilhafte Ausführung umfaßt ein Verfahren zur Durchführung der Validationsphase.

Eine weitere vorteilhafte Ausführung umfaßt ein Verfahren, bei dem die Verfeinerungsphase der Restaurationssequenz durchgeführt wird, indem ein binärer Suchvorgang durchgeführt wird, um eine kürzeste Teilsequenz zu erkennen, die alle Fehler in der Zielfehlerliste erkennt.

Eine weitere vorteilhafte Ausführung umfaßt ein Verfahren zur Durchführung der Verfeinerungsphase.

Ein weiterer Aspekt der vorliegenden Erfindung betrifft ein beschleunigtes Restaurationsverfahren einer Sequenz von

Testvektoren und umfaßt die nachfolgenden Schritte:
Identifikation von Testvektoren, einer Fehlerliste mit Fehlern, die unter Verwendung der Testvektoren erkannt werden können, und von Detektionszeiten für die Fehler;
Auswahl von Fehlern, die der Zielfehlerliste zugeordnet werden, falls die Fehler existieren;
Durchführung einer überlappenden Validation, so daß wenn zwei Fehler in der Zielfehlerliste restaurierte Sequenzen haben, die sich überlappen, die beiden Fehler vermischt werden, so daß sie einen Zielfehler bilden;
Durchführung einer überlappenden Verfeinerung, falls ein Segment existiert, das den einen Zielfehler erkennt; und
Wiederholung solange Zielfehler existieren.

Eine weitere vorteilhafte Ausführung der vorliegenden Erfindung umfaßt ein Verfahren, bei dem während der Durchführung einer überlappenden Validation, falls restaurierte Sequenzen von zwei Fehlern gemeinsame Vektoren haben, die Restaurationsverfahren der beiden Vektoren überlappt werden.

Eine weitere vorteilhafte Ausführung der vorliegenden Erfindung umfaßt ein Verfahren, bei dem solange wenigstens ein Fehler der Zielfehlerliste während dem überlappenden Validationsverfahren unerkannt ist, neue Zielfehler, die nicht in der Zielfehlerliste enthalten sind, zu der Zielfehlerliste hinzugefügt werden.

Eine weitere vorteilhafte Ausführung der vorliegenden Erfindung umfaßt ein Verfahren, bei dem während der Durchführung einer überlappenden Verfeinerung ein Segment zwischen einer low Sequenz, die alle Fehler in der Zielfehlerliste identifiziert, und einer high Teilsequenz identifiziert, innerhalb der wenigstens ein Fehler der Zielfehlerliste nicht identifiziert wird.

Eine weitere vorteilhafte Ausführung der vorliegenden Erfindung umfaßt ein Verfahren, bei dem solange wenigstens ein Fehler der Zielfehler während der überlappenden Verfeinerungsphase unerkannt ist, neue Fehler zu der Zielfehlerliste hinzugefügt werden, falls neue Fehler, die nicht in der Zielfehlerliste sind, identifiziert werden.

Weitere vorteilhafte Ausführungen der vorliegenden Erfindung umfassen ein Verfahren zur Durchführung einer überlappenden Validation, einer überlappenden Verfeinerung, und Aktualisierung von Zielfehlerlisten und den Werten HIGH und LOW.

KURZE BESCHREIBUNG DER ZEICHNUNGEN

Die vorstehenden Aufgaben, Merkmale und Vorteile der vorliegenden Erfindung werden nachfolgend ohne jede Beschränkung anhand von Ausführungsbeispielen detailliert beschrieben, die in schematischen Zeichnungen dargestellt sind. Hierzu zeigt:

Fig. 1 ein Beispiel einer Testmenge und einer entsprechenden Fehlermenge;
Fig. 2 ein Blockdiagramm eines Kompaktierungsverfahrens mit Vektorrestauration;
Fig. 3 eine graphische Darstellung einer Ausführung eines Zweiphasen-Vektorrestaurationsschemas;
Fig. 4 eine Pseudo-Code-Implementation für eine Ausführung eines Zweiphasen-Vektorrestaurations-Verfahrens;
Fig. 5 einen Vergleich der herkömmlichen linearen Vektorrestauration mit der Zweiphasen-Restauration;
Fig. 6 ein Blockdiagramm eines beschleunigten Zweiphasen-Vektorrestaurationsschemas;
Fig. 7 eine Pseudo-Code-Implementation einer Ausführung eines beschleunigten Zweiphasen-Vektorrestaurations-Verfahrens;
Fig. 8 ein Blockdiagramm eines überlappenden Validationsschemas;
Fig. 9 eine Pseudo-Code-Implementation einer Ausführung des überlappenden Validationsverfahrens;
Fig. 10 eine Pseudo-Code-Implementation einer Ausführung eines Algorithmus, der Bereichsinformationen (range information) (LOW und HIGH arrays) aktualisiert;
Fig. 11 eine Pseudo-Code-Implementation einer Ausführung eines Verfahrens das die Zielfehlerliste aktualisiert;
Fig. 12 einen Pseudo-Code einer Ausführung eines Verfahrens, das die Existenz von Segmenten überprüft;
Fig. 13 ein Blockdiagramm eines Verfahrens zur überlappenden Verfeinerung;
Fig. 14 eine Pseudo-Code-Implementation einer Ausführung des überlappenden Verfeinerungsverfahrens;
Fig. 15 ein Beispiel für eine beschleunigte Zweiphasen-Restauration;
Fig. 16 Tabelle 1 mit den Resultaten für ISCAS-Schaltkreise; und
Fig. 17 Tabelle 2 mit Resultaten für verschiedene industrielle Designs, auf die die vorliegende Erfindung angewendet wurde.

DETAILLIERTE BESCHREIBUNG DER ERFINDUNG

Die vorliegende Erfindung stellt ein neues Verfahren zur Vektorrestauration bereit. Ein neues Zweiphasen-Restaurationsschema wird nachfolgend detailliert beschrieben, das für große Designs geeignet ist. Das erfindungsgemäße Verfahren ist der herkömmlichen linearen Vektorrestauration weit überlegen. Die nachstehenden Techniken können verwendet werden, um Laufzeiten von Anwendungen wie statische Kompaktierung und Fehlerdiagnose zu verbessern. Aus Experimenten geht hervor, daß mittels dem erfindungsgemäßen Restaurationsverfahren eine statische TestSequenz-Kompaktierung große industrielle Designs verarbeiten kann, die mit herkömmlichen linearen Vektorrestaurationsverfahren nicht bewältigt werden konnten. Herkömmliche lineare Vektorrestaurationsverfahren sind in I. Pomeranz und S. M. Reddy "Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential circuits", Proceedings Int. Conf. on Computer Design, S. 360–365, 1997, University of Iowa, August 1997, beschrieben.

Zweiphasen-Vektorrestauration

Das erfindungsgemäße Restaurationsverfahren besteht aus zwei Phasen:
Validation und Verfeinerung. In der Validationsphase wird eine hinreichende Teilsequenz von Vektoren schnell identifiziert. Eine derartige Teilsequenz enthält in jedem Fall auch eine kürzere Teilsequenz von Vektoren, die alle Zielfehler er-

kennen. Diese Teilsequenz ist größer als nötig und wird als validiertes Segment (validated segment) bezeichnet.

Die Verfeinerungsphase gleicht das validierte Segment ab und findet die kürzeste Teilsequenz, die alle Zielfehler erkennt.

Das erfindungsgemäße Restaurationsverfahren ermöglicht es, daß Fehler von nichtüberlappenden Teilsequenzen erkannt werden. Die Restauration von Fehlern f_3 und f_5 in dem Beispiel von **Fig.** 1 zeigt einen derartigen Fall. Die vollständige Testmenge kann als Teilsequenz vorgeschlagen werden, die benötigt wird, um die Fehler f_3 und f_5 zu entdecken. Teilsequenzen, die die Fehler f_3 und f_5 erkennen, haben jedoch keine gemeinsamen Vektoren. Wie vorstehend gesagt, erkennt Teilsequenz $v_{17}, \ldots, v_{20}$ den Fehler f_5 und Teilsequenz $v_1, \ldots, v_{12}$ erkennt den Fehler f_3. Diese Teilsequenzen haben keinen gemeinsamen Vektor.

Das erfindungsgemäße Restaurationsverfahren erzeugt anstelle einer einzigen Teilsequenz zwei Teilsequenzen. Es ist klar, daß diese Teilsequenzen auf beliebige Weise angeordnet sein können, um die Fehler f_3 und f_5 zu erkennen. Derartige unabhängige Teilsequenzen werden als Segmente bezeichnet. Das Restaurationsverfahren der vorliegenden Erfindung restauriert nichtüberlappende Segmente, die beliebig angeordnet sein können, um alle Zielfehler zu erkennen.

Fig. 3 zeigt eine Ausführung eines erfindungsgemäßen Zweiphasen-Restaurationsschemas anhand dessen ein dem erfindungsgemäßen Zweiphasen-Restaurationsverfahren zugrundeliegendes Schlüsselkonzept erläutert werden kann. Der aktuelle Zielfehler ist f. Seine Detektionszeit ist mit D[f] gekennzeichnet. Die Label low, opt, high, base und last nehmen Werte zwischen 1 und n an (ursprüngliche Testmenge besteht aus Vektoren $v_1, \ldots, v_n$). Die Sequenz ResSeq ($v_{base}, \ldots, v_{last}$) ist restauriert für frühere Ziele und erkennt das aktuelle Zielfehler f nicht.

Das Restaurationsverfahren für f beginnt bei Vektor v_{base}. Die Validationsphase identifiziert eine Teilsequenz $v_{low}, \ldots, v_{last}$, die den Zielfehler erkennt. Die Restaurationsphase legt auch fest, daß die Teilsequenz $V_{high}, \ldots, v_{last}$ den Zielfehler nicht erkennt. Wesentlich ist, daß die kürzeste Teilsequenz, die das Ziel erkennt, bei einem Vektor zwischen v_{low} und v_{high} beginnt.

Die Verfeinerungsphase identifiziert die Teilsequenz $v_{opt}, \ldots, v_{base}$ als die kürzeste Teilsequenz, die zu der restaurierten Teilsequenz ResSeq hinzugefügt ist. Hier gilt, low $\leq$ opt $\leq$ high.

Eine erfindungsgemäße Ausführung eines Zweiphasen-Restaurationsverfahrens ist von dem in **Fig.** 4 dargestellten Pseudo-Code beschrieben. Das Restaurationsverfahren wiederholt Validations- und Verfeinerungsphasen bis alle Fehler in der Eingangsfehlerliste F_u von der restaurierten Sequenz ResSeq erkannt sind. Während jeder Iteration werden nur Fehler berücksichtigt, die von der aktuell restaurierten Sequenz ResSeq nicht erkannt werden. Von diesen unerkannten Fehlern werden die Fehler als Zielfehler ausgewählt, die die höchsten Detektionszeiten haben. Diese Fehler werden in die Zielfehlerliste F_T geschrieben. Die nächsten beiden Abschnitte beschreiben die Validations- und Verfeinerungsphasen.

Validation

Bei einer gegebenen Zielfehlerliste F_T bestimmt die Validationsphase die Werte der Variablen low und high, die das validierte Segment begrenzen. Diese Phase kann entweder eine aus früheren Restaurationsphasen restaurierte Sequenz ResSeq übernommen haben, oder kann eine neue Sequenz beginnen. Eine Variable base wird initialisiert, so daß sie das Minimum einer Detektionszeit eines Fehler in F_T ist, und daß sie der Index des ersten Vektors in der schon restaurierten Sequenz ResSeq ist. Das Restaurationsverfahren dehnt sich auf ResSeq aus, um Fehler in F_T zu erkennen.

Die while-Schleife (**Fig.** 4, Zeile 6) ruft den Fehlersimulator (**Fig.** 4, Zeile 8) mehrmals auf. Wenn ResSeq Vektoren hat, dann verifiziert der Fehlersimulator, ob F_T schon von ResSeq erkannt wurde. Falls ein oder mehrere Fehler von F_T nicht erkannt sind, dann werden zusätzliche Vektoren zu ResSeq hinzugefügt (**Fig.** 4, Zeilen 10 und 11). Dieses Verfahren wird wiederholt, bis genügend Vektoren hinzugefügt sind, um F_T zu erkennen.

Es kann gezeigt werden, daß der Validationsschritt den Fehlersimulator meistens O(log k) mal aufruft, wenn k Vektoren zu ResSeq hinzugefügt werden, um F_T zu erkennen. Im schlechtesten Fall simuliert die Validationsphase nur $2^{\lfloor \log 2k \rfloor + 1}$ Vektoren. Dabei ist die Simulation von Vektoren in ResSeq nicht enthalten.

Der Arbeitsablauf der Validationsphase kann anhand dem Beispiel von **Fig.** 1 aufgezeigt werden, indem betrachtet wird, wie eine Sequenz für den Fehler f_5 restauriert wird. Da f_5 der erste restaurierte Fehler ist, hat ResSeq keine Vektoren. Daher werden dem Wert $D[f_5] = 20$ base und low zugeordnet. Während der ersten Iteration der while-Schleife (**Fig.** 4, Zeile 6) ermittelt der Fehlersimulator, daß f_5 von der Sequenz, die aus einem Vektor v_{20} besteht, nicht erkannt wird. Da f_5 nicht erkannt wird, wird die Variable low aktualisiert und erhält den Wert $20-2^0 = 19$. In der nächsten Iteration ermittelt der Fehlersimulator, daß f_5 von der Sequenz v_{19}, v_{20} nicht erkannt wird. Daher wird die Variable high aktualisiert und erhält den Wert 19. Die Variable low wird ebenso aktualisiert und erhält den Wert $20-2^1 = 18$.

Bei der Fortführung dieses Verfahrens wird f_5 erkannt, wenn low $= 20-2^2 = 16$. Das kommt daher, daß die Sequenz $v_{16}, \ldots, v_{20}$ die Fehler von einem unbekannten ursprünglichen Zustand detektiert. Das Hinzufügen des Vektors v_{16} kann die Erkennbarkeit des Fehlers nicht ändern. An diesem Punkt ist high = 18. Dies markiert das Ende der Validationsphase. Es wurden vier Iterationen der while-Schleife benötigt. Das validierte Segment besteht aus den Vektoren $v_{16}, \ldots, v_{20}$. Dieses Segment enthält mehr Vektoren als notwendig, um f_5 zu erkennen.

Verfeinerung

Nach der Validationsphase werden alle Fehler in der Zielfehlermenge F_T als unerkannt markiert (**Fig.** 4, Zeile 13). Die Verfeinerungsphase identifiziert die kürzeste Teilsequenz in dem validierten Segment, das F_T erkennt.

Die while-Schleife (**Fig.** 4, Zeile 14) in der Verfeinerungsphase ruft auch einige Male den Fehlersimulator auf (**Fig.** 4, Zeile 17). Eine einfache binäre Such-Prozedur wird verwendet, um die kürzeste Teilsequenz, die alle Fehler in F_T erkennt, zu vergrößern. Der Fehlersimulator wird von der Verfeinerungsphase höchstens O(log (high - low) mal aufgerufen. Falls die Länge der restaurierten Sequenz k ist, dann werden schlechtestenfalls O(log k) Aufrufe des Fehlersimulators benötigt. Auch die Verfeinerungsphase kann schlechtestenfalls die Simulation von $2k \cdot \log k$ Vektoren benötigen.

Hierin sind wiederum nicht die Vektorsimulationen in ResSeq enthalten.

Die Verfeinerungsphase für f_5 (**Fig.** 1) schließt eine binäre Suche zwischen low = 16 und high = 18 nach der kürzesten Teilsequenz ein, die f_5 erkennt. Die erste Sequenz, die berücksichtigt wird, beginnt bei Vektor v_{17}. Das kommt daher, weil (low + high)/2 = 17 ist. Die Sequenz $v_{17}, \ldots, v_{20}$ erkennt f_5. Daher wird low aktualisiert und erhält den Wert 17. Da low = high + 1 ist schließt die while-Schleife von Zeile 14 und beendet die Verfeinerungsphase. Die kürzeste bekannte Teilsequenz ist $v_{17}, \ldots, v_{20}$.

Vergleich mit der linearen Vektorrestauration

Zunächst wird die Komplexität des linearen Vektorrestaurations(LVR)-Verfahrens analysiert. Bei der Vektorrestauration für Fehler f_5, wird die erste Sequenz mit der Länge 1 (v_{20}) berücksichtigt, gefolgt von einer Sequenz mit der Länge 2 (v_{19}, v_{20}), und dieses Verfahren wird bis zu der Sequenz $v_{17}, \ldots, v_{20}$ fortgeführt. Bei jeder Iteration wird ein Vektor hinzugefügt. Demzufolge ist die Gesamtzahl der während der Restauration simulierten Vektoren 1 + 2 + 3 + 4. Hier ist die Länge der restaurierten Teilsequenz gleich 4 und vier Schritte werden benötigt, um die Sequenz wiederzufinden. Die vorstehende Berechnung kann für eine restaurierte Teilsequenz der Länge k verallgemeinert werden:

- Anzahl von simulierten Vektoren während der Restauration = $1 + 2 + \ldots + k = k(k + 1)/2$.
- Anzahl von Iterationen, um die Teilsequenz zu restaurieren = k (kennzeichnet auch die Anzahl der Aufrufe des Fehlersimulators).

Fig. 5 zeigt die verschiedenen Schritte innerhalb der LVR und der Zweiphasenrestauration, während der Vektor f_5 des Beispiels von **Fig.** 1 restauriert wird.

Die letzte von LVR und 2ϕ restaurierte Sequenz sind identisch. Bei einer restaurierten Sequenz mit der Länge k ergibt eine sorgfältige Analyse des Verfahrens von **Fig.** 4:

- Die Validationphase benötigt eine Simulation von $2^{\lceil \log 2k \rceil + 1}$ Vektoren
- Die Verfeinerungsphase könnte im schlechtesten Fall eine Simulation von $2k \cdot \log_2 k$ Vektoren benötigen.
- Das Verfahren von **Fig.** 4 ruft den Fehlersimulator höchstens $2 \log_2 k + 1$ mal auf.

Die Signifikanz der obigen Analyse wird bei der Betrachtung einer typischen Anzahl für k offensichtlich. Wenn die Sequenz ResSeq 1000 Vektoren hat, und das Restaurationsverfahren für ein Ziel eine Addition von 100 Vektoren zu ResSeq benötigt, dann ist der Wert von k gleich 100. Bei jeden Aufruf des Fehlersimulators werden Vektoren aus ResSeq simuliert. Wenn die Anzahl der Vektoren, die während der Restauration simuliert werden, V_{LVR} bei der linearen Restauration und $V_{2\phi}$ bei dem Zweiphasenverfahren ist, dann gilt

$$V_{LVR} = 1000 \cdot (\#\text{ Aufrufe Fehlersimulator}) + 100 \cdot (100 + 1)/2 = 1000 \cdot 100 + 5050 = 105{,}050$$
$$V_{2\phi} = 1000 \cdot (\#\text{ Aufrufe Fehlersimulator}) + (256 + 2 \cdot 100 \cdot 7) = 1000 \cdot (2 \cdot 7 + 1) + (256 + 2 \cdot 100 \cdot 7) = 16{,}656$$

Verglichen mit der linearen Restauration benötigt die Zweiphasen-Restauration fast um eine Größenordnung weniger Simulationen. Demzufolge wird von der Zweiphasen-Restauration erzielt eine deutliche Geschwindigkeitserhöhung.

Beschleunigte Zweiphasen-Restauration

Die im Zusammenhang mit **Fig.** 4 beschriebene Zweiphasen-Restauration berücksichtigt ein neues Ziel nur nach der Restauration der kürzesten Sequenz für ein aktuelles Ziel. Typischerweise überlappen sich restaurierte Sequenzen aufeinanderfolgender Ziele. Es kann daher sein, daß es nicht nötig ist, die kürzeste Sequenz für das aktuelle Ziel zu finden. In vielen Fällen ist es möglich, das nächste Ziel zu identifizieren, selbst bevor die Vektorrestauration für das aktuelle Ziel abgeschlossen ist.

Das beschleunigte Zweiphasen-Restaurationsverfahren ist ein weiterer Aspekt der vorliegenden Erfindung. Bei diesem Verfahren werden neue Ziele entweder in der Validationsphase oder der Verfeinerungsphase identifiziert. Wenn neue Ziele identifiziert werden, wird das Restaurationsverfahren für das aktuelle Ziel mit dem Restaurationsverfahren des neuen Ziels überlappt. Wie nachstehend dargelegt, wird mit dem Überlappen der Vektorrestauration bei dem Zweiphasen-Verfahren eine weitere deutliche Beschleunigung des Restaurations-Verfahrens erzielt. Bei dem Zweiphasen-Verfahren und dem beschleunigten Zweiphasen-Verfahren sind die restaurierten Segmente identisch.

Fig. 6 zeigt den Ablauf des beschleunigten Restaurationsverfahrens. Wie bei einem grundlegenden Vektorrestaurationsverfahren wird eine Menge von Testvektoren eingegeben, eine Liste von zu detektierenden Fehlern wird eingegeben, und eine Liste von Detektionszeiten für die Fehler von 6.01 wird eingegeben. Anders als bei dem grundlegenden Zweiphasen-Restaurationsverfahren, das vorstehend beschrieben ist, wird bei dem beschleunigten Verfahren die Validation und Verfeinerung einer Vielzahl von Zielfehlern überlappt. Der Restaurationsabschnitt des beschleunigten Verfahrens besteht insbesondere aus drei Phasen: Überlappte Validation **6.05**, Segmentexistenzbedingung **6.06** und überlappte Verfeinerungsphase **6.07**.

Nachfolgend wird ein Schlüsselkonzept beschrieben, das dem beschleunigten Verfahren zugrundeliegt. Falls eine Teilsequenz, die nach der Validationsphase erhalten ist, kein Segment für Zielfehler enthält, dann ist eine Verfeinerungsphase nicht notwendig. Die beschleunigte Version überlappt die Validation für eine Vielzahl Fehler bis eine Teilsequenz identifiziert ist, die ein Segment enthält. Die Verfeinerungsphase **6.07** extrahiert ein Segment für Zielfehler. Diese Phase überlappt wiederum Verfeinerungsphasen mehrerer Zielfehler.

Fig. 7 zeigt die Schritte des beschleunigten Verfahrens. Das beschleunigte Verfahren wiederholt ähnlich wie das grundlegende zweiphasige Verfahren auch Validations- und Verfeinerungsphasen, bis alle Fehler in der Eingangsfehler-

liste F_u von der restaurierten Sequenz ResSeq erkannt sind. Das beschleunigte Verfahren berechnet auch die Zielfehlerliste F_T zu Beginn jeder Iteration. Es gibt jedoch einige grundlegende Unterschiede zwischen den grundlegenden Restaurationsverfahren und dessen beschleunigte Version:

(1) Das beschleunigte Verfahren modifiziert die Zielfehlerliste in der Validations- oder Verfeinerungsphase,
(2) das beschleunigte Verfahren führt eine Verfeinerung nur aus, nachdem die Validationsphase eine Sequenz erzeugt hat, die ein Segment für Zielfehler enthält, und
(3) jede Verfeinerungsphase der while-Schleife (**Fig.** 6, Zeile 8) in dem beschleunigten Verfahren erzeugt genau ein Segment.

Das ist anders als bei dem grundlegenden Verfahren, bei dem mehrere Verfeinerungsphasen benötigt werden können, um ein Segment zu erzeugen, das Zielfehler detektiert.

Das beschleunigte Verfahren speichert auch mehr Informationen über den Bereich von Vektoren, die ein Ziel erkennen oder nicht erkennen. Diese Information ist während der Fehlersimulation einer Sequenz und Zielfehlern leicht erhältlich. Bei dem Verfahren von **Fig.** 4 ermittelt die Validationsphase Werte von Variablen low und high, die die folgenden Eigenschaften erfüllen: (1) Teilsequenz von $v_{low}, \ldots, v_{last}$, erkennt alle Zielfehler F_T, aber (2) Teilsequenz $v_{high}, \ldots, v_{last}$ erkennt nicht wenigstens einen Fehler in F_T.

Das beschleunigte Verfahren verwendet Arrays LOW und HIGH, um den Informationsbereich jedes Fehlers aufzuzeichnen. Für einen Fehler f gilt unter der Annahme eines Fehlers f mit LOW[f] = i und HIGH[f] = j, wobei die Werte i und j jeweils die nachstehenden Eigenschaften erfüllen:

(1) Sequenz $v_i, \ldots, v_{last}$, erkennt den Fehler, aber
(2) Sequenz $v_j, \ldots, v_{last}$ erkennt den Fehler nicht. Die Bereichsinformation jedes Fehlers wird während der überlappten Validation oder Verfeinerung kontinuierlich aktualisiert. Die überlappte Validation und Verfeinerung wird nachfolgend beschrieben.

Überlappte Validation

Unter der Annahme, daß f_x das aktuelle Ziel ist, detektiert bei der Validationsphase des Zweiphasen-Restaurationsverfahrens von **Fig.** 4 eine zwischenliegende Sequenz $v_{low}, \ldots, v_{last}$ (siehe **Fig.** 3) nicht f_x, und diese Sequenz detektiert auch keinen anderen Fehler f_y, der eine Detektionszeit $D[f_y] \geq$ low hat. In einem derartigen Fall überlappen die restaurierten Sequenzen der Fehler f_x und f_y. Beide Sequenzen enthalten insbesondere Vektoren $v_{low}, \ldots, v_{D[fy]}$. Daher kann die Validation der Fehler f_x und f_y überlappt sein.

In dem Beispiel von **Fig.** 1 wird während der Validation für f_3 die Sequenz $v_4, \ldots, v_{12}$ als ein möglicher Kandidat berücksichtigt. Die falsche Simulation deckt auf, daß f_2 und f_3 nicht erkannt sind. Die Detektionszeit von f_2 ist 5. Daher wird eine beliebige restaurierte Sequenz, die f_2 detektiert mit der restaurierten Sequenz für den Fehler f_3 wenigstens die Vektoren v_4 und v_5 teilen. Daher kann die für f_3 restaurierte Sequenz nicht alleine ein Segment bilden. Um die Segmentrestauration zu beschleunigen, können beide Fehler als ein Ziel $F_T = (f_2, f_3)$ vermischt werden. Die Validationsphase kann die Restauration für die Fehler f_2 und f_3 überlappen.

Der erfindungsgemäßen überlappten Validation liegt ein Schlüsselkonzept zugrunde, das nachstehend beschrieben wird. Falls restaurierte Sequenzen zweier Fehler gemeinsame Vektoren haben, kann das Restaurationsverfahren für die beiden Fehler überlappt werden. Hierdurch wird ein bedeutender Rechenvorteil erzielt.

Fig. 8 zeigt das überlappte Validationsverfahren. Zu Beginn des Verfahrens wird eine Zielfehlerliste eingegeben und partielle Segmente von Testvektoren werden eingegeben, und die Detektionszeiten von Fehlern werden in **8.01** eingegeben. In **8.02** werden Vektoren zu dem partiellen Segment hinzugefügt. In **8.03** wird ermittelt, ob das Ziel detektiert ist. Wenn das Ziel erkannt ist, wird nachdem das validierte Segment in **8.04** ausgegeben ist, der Algorithmus verlassen, anderenfalls wird die Zielfehlerliste in **8.05** aktualisiert, und das Verfahren mit **8.02** fortgeführt.

Wenn der gestrichelte Block **8.05** von **Fig.** 8, der die Zielfehlerliste aktualisiert, ignoriert wird, dann beschreibt das Flußdiagramm die Validationsphase des grundlegenden Verfahrens. Bei der überlappten Validation wird ein zusätzlicher Prozeß durchgeführt, wenn ein oder mehrere Fehler in der Zielfehlerliste F_T nicht erkannt sind. Dieser Prozeß ermittelt, ob neue Ziele identifiziert wurden. Falls ein neues Ziel gefunden wird, dann wird die Validationsphase fortgeführt, indem neue Ziele in die aktuelle Zielfehlerliste aufgenommen werden.

Fig. 9 zeigt den detaillierten Algorithmus für die überlappte Validationsphase. Während jeder Iteration der while-Schleife (**Fig.** 9, Zeile 3) wird eine neue Sequenz Seq vorgeschlagen (**Fig.** 9, Zeile 5). Es wird auch eine neue Fehlerliste F_s gebildet (**Fig.** 9, Zeile 4). Diese Menge enthält alle unerkannten Fehler, die Detektionszeiten entsprechend den Vektoren in Seq haben. Für die Aktualisierung des Informationsbereichs für die Fehler in F_s wird eine falsche Simulation der Sequenz Seq verwendet Man betrachtet beispielsweise einen Fehler F aus F_s, der von der falschen Simulation erkannt wird. Der Index des ersten Vektors in Seq sei Index. Falls LOW[f] < Index ist dann wird der Wert von LOW von [f] aktualisiert und LOW erhält den Wert Index. Das wird durchgeführt, weil die Fehlersimulation festgelegt hat, daß Fehler f von der Sequenz Seq erkannt wird. Falls die falsche Simulation den Fehler f nicht erkannte, kann der Wert von HIGH[f] aktualisiert werden. Wenn beispielsweise HIGH[f] > Index ist, dann wird der Wert von HIGH[f] aktualisiert und erhält den Wert Index. **Fig.** 10 zeigt das Verfahren für die Aktualisierung der Bereichsinformationen.

Wenn eine oder mehrere Fehler in Fr von der Seq nicht erkannt werden, können neue Ziele identifiziert werden. Dieser Schritt existiert nicht in dem grundlegenden Verfahren. Unter Verwendung des Verfahrens UPDATE_TARGETS, das in **Fig.** 11 dargestellt ist, werden neue Ziele identifiziert. Unter Verwendung der Bereichsinformationen der Fehler in F_T wird der Index high zuerst derart identifiziert, daß die Teilsequenz $v_{high}, \ldots, v_{last}$ nicht wenigstens einen Fehler in F_T erkennt. Unentdeckte Fehler, die nicht Teil der Zielfehlerliste F_T sind, (**Fig.** 11, Zeile 3) werden in Abhängigkeit ihrer De-

tektionszeiten in absteigender Reihenfolge berücksichtigt. Man nehme an, f_p sei ein derartiger unerkannter Fehler.

Unter Berücksichtigung des Vektors $v_{D[fp]}$, falls high $\leq D[f_p] + 1$ ist, wird die restaurierte Sequenz für F_T tatsächlich den Vektor $v_{D[fp]}$ enthalten. Wenn auch $HIGH[f_p] \leq D[f_p]$ ist, dann wird die für f_p restaurierte Sequenz auch den Vektor $v_{D[fp]}$ enthalten. Das kommt daher, daß es bekannt ist, daß die Sequenz $v_{HIGH[fp]}, \ldots, v_{last}$ f_p nicht detektieren können. Falls der Vektor $v_{D[fp]}$ restaurierten Sequenzen für Fehler in F_T und der Fehler f_p gemeinsam ist, können das Restaurationsverfahren für diese Fehler überlappt werden, indem f_p in F_T enthalten ist.

Es kann sein, daß F_T Fehler enthält, die von der Sequenz $v_{high}, \ldots, v_{last}$ erkannt werden. Man betrachte beispielsweise einen Fehler f aus F_T. Wenn $LOW[f] \geq max$ ist, dann definiert die Sequenz $v_{high}, \ldots, v_{last}$ mit Sicherheit den Fehler f. Zeile 10 in **Fig.** 11 identifiziert derartige Fehler. Diese Fehler können als erkannt gekennzeichnet werden, und sie müssen nicht nochmals bei dem Restaurationsverfahren berücksichtigt werden. Daher werden sie aus der Fehlerliste F_u und der Zielliste F_T entfernt. Falls f_p als neues Ziel aufgenommen wird, dann wird der Wert von high (**Fig.** 11, Zeile 4) neu errechnet. Das Aktualisierungsverfahren hält bei dem ersten Fehler f_p an, der kein neues Ziel ist. Es wird so verfahren, da restaurierte Sequenzen für verbleibende, nicht erkannte Fehler nicht mit dem Segment für die Zielfehler F_T überlappen.

Segmentexistenzprüfung

Die Sequenz, die nach der Validation erhalten ist, kann ein Segment enthalten oder kann kein Segment enthalten. Falls kein Segment existiert, dann ist keine Verfeinerungsphase nötig. Das Verfahren von **Fig.** 12 ermittelt, ob eine Sequenz ein Segment enthält. Die Sequenz ResSeq erkennt keine Fehler in F_T. Die Bereichsinformation von Fehlern in F_T wird verwendet, um den Vert der Variablen low (**Fig.** 12, Zeile 2) zu ermitteln. Die Sequenz $Seq = \{v_{low}, \ldots, v_{base}\} \cup ResSeq$ ermittelt alle Fehler in F_T. Diese Prozedur nimmt an, daß eine Fehlersimulation für Seq bereits durchgeführt wurde.

SEGMENT FOUND ist eine Ausführung eines erfindungsgemäßen Verfahrens, das ermittelt, ob eine Teilsequenz $v_{low}, \ldots, v_{base}$ ein Segment enthält, das F_T erkennt. In dieser Phase ist es nicht nötig, das exakte Segment für F_T zu identifizieren. Es genügt, die Existenz einer Teilsequenz in Seq festzulegen, die den nachstehenden beiden Eigenschaften genügt:

(1) Teilsequenz erkennt F_T, und
(2) kein unerkannter Fehler in $F_u - F_T$ hat eine Detektionszeit, die einem Vektor in der Teilsequenz entspricht.

Der erste Schritt ist die Berechnung von F_c. Diese Menge enthält alle unerkannten Fehler mit Detektionszeiten, die Vektoren in Seq entsprechen. Falls F_c keine Fehler hat, dann enthält Seq ein Segment (**Fig.** 12, Zeile 6). Andererseits müssen Vektoren, die zu restaurierten Sequenzen für Fehler in F_c gehören, ausgeschlossen werden. Man betrachte einen Fehler f aus F_c. Da $HIGH[f] \leq low$ ist, wird der Fehler f von Seq nicht erkannt. Wichtiger ist, daß eine restaurierte Sequenz für den Fehler f mit Seq überlappt. Die Überlappung kann die Vektoren $v_{low}, \ldots, v_{D[f]}$ betreffen.

Falls daher ein Segment für F_T existiert, dann muß es in der Teilsequenz von $v_{D[f]+1}, \ldots, v_{base}$ enthalten sein. Zeile 8 in **Fig.** 12 ermittelt den Index low des ersten Vektors, der nicht zur restaurierten Sequenz für irgendeinen Fehler in F_c gehört.

Auf der Grundlage der Fehler in F_c wird eine kürzere Sequenz Seq (**Fig.** 12, Zeile 9) vorgeschlagen. Die neue Sequenz Seq kann nicht alle Fehler in F_T erkennen. Dies wird von einer falschen Simulation ermittelt. Wenn dies der Fall ist (**Fig.** 12, Zeilen 16–17), dann enthält Seq nicht ein Segment für F_T. Falls die Seq alle Fehler in F_T erkennt (**Fig.** 12, Zeile 14), dann ist nicht klar, ob ein Segment für F_T existiert oder nicht. Es gibt jedoch jetzt eine kürzere zu prüfende Sequenz als verglichen mit der ursprünglichen Sequenz zu Beginn der while-Schleife (**Fig.** 12, Zeile 3). Das Verfahren wird für die kürzere Sequenz wiederholt. Dieses Verfahren wird nachstehend unter Verwendung eines Beispiels erläutert.

Überlappte Verfeinerung

In dem Beispiel von **Fig.** 1 ist nach der Validationsphase des grundlegenden Verfahrens (**Fig.** 4) für den Fehler f_3 die Zielfehlerliste $F_T = (f_3, f_2)$, high = 4 und low = 1. Auf der Basis dieser ursprünglichen Werte ist die erste Sequenz, die von dem Fehlersimulator verarbeitet wird, $v_2, \ldots, v_{12}$. Diese Sequenz erkennt beide Fehler in F_T. Der Fehler f_1 wird jedoch nicht erkannt. Seine Detektionszeit $D[f_1] = 3$ entspricht dem Vektor v_3, der bereits in der Sequenz $v_2, \ldots, v_{12}$ enthalten ist. Daher enthält die für f1 restaurierte Sequenz auch Vektoren v_2 und v_3 und eine weitere Verfeinerung von $v_2, \ldots, v_{12}$ ist nicht nötig. Verfeinerungsphasen von Fehler f_1 und Fehlern in F_T sollten überlappt werden, um das Segment zu finden.

Ein grundlegendes Konzept der überlappten Verfeinerung ist ein Segment zu finden. Wie bei der überlappten Validation werden neue Ziele identifiziert, falls ein oder mehrere Fehler in F_T nicht erkannt werden. Falls ein neues Ziel gefunden ist, wird die Verfeinerungsphase fortgeführt, indem ein neues Ziel in F_T eingefügt wird. Anders als bei den grundlegenden Verfahren von **Fig.** 4, stellt die überlappte Verfeinerungsphase

(1) immer ein Segment her, und
(2) null Fehler können zu F_T während der Verfeinerungsphase hinzugefügt werden.

Eine erfindungsgemäße Ausführung des überlappten Verfeinerungsverfahrens ist in **Fig.** 13 dargestellt. Zu Beginn des Verfahrens wird ein validiertes Segment und eine Liste von Zielfehlern zusammen mit ihren Detektionszeiten eingegeben. Das eingegebene validierte Segment wird in **13.02** beschnitten (pruned). In **13.03** ermittelt das Verfahren, ob das Ziel erkannt ist. Wenn das Ziel nicht erkannt ist, wird die Fehlerliste in **13.06** aktualisiert, und die Beschneidung (pruning) des validierten Segments in **1.302** wird fortgesetzt. Falls das Ziel erkannt ist, ermittelt Bedingung **13.04**, ob weitere Verfeinerung möglich ist. Wenn die Antwort "yes" ist, wird die Beschneidung (pruning) des validierten Segments in **13.02** fortgesetzt, anderenfalls wird die Ausführung beendet und das gefundene Segment ausgegeben.

Falls der gestrichelte Block **13.06** weggelassen wird, dann zeigt das Flußdiagramm die Verfeinerungsphase des grund-

legenden Verfahrens.

Eine erfindungsgemäße Ausführung der detaillierten Schritte des Verfahrens für die überlappte Verfeinerungsphase ist in **Fig.** 14 dargestellt. Ursprünglich enthält ResSeq keine Fehler in F_T. Die Bereichsinformation von Fehlern in F_T wird erneut benutzt, um den ersten Vektor v_{low}, der Sequenz zu finden, die Fehler in F_T erkennt (**Fig.** 14, Zeile 2). Alle Fehler in F_T werden von der Teilsequenz $v_{low}, \ldots, v_{base}$ erkannt. Die Bereichsinformation wird auch verwendet, um den Index high zu identifizieren, so daß wenigstens ein Fehler in F_T nicht erkannt wird. Diese Sequenz ist $v_{high}, \ldots, v_{base}$. Ein Segment für F_T beginnt irgendwo zwischen low und high. Die überlappte Verfeinerung identifiziert dieses Segment. Dieses Segment erkennt alle Fehler in F_T, aber es kann auch andere Fehler erkennen. Die zusätzlichen in dem Segment erkannten Fehler können neue Ziele werden, die in F_T enthalten sind.

Während jeder Iteration der while-Schleife (**Fig.** 14, Zeile 4) wird eine neue Sequenz Seq vorgeschlagen (**Fig.** 14, Zeile 6). Eine neue Fehlerliste F_s wird gebildet, die unerkannte Fehler enthält, mit Detektionszeiten, die Vektoren in Seq entsprechen. Fehlersimulation der Sequenz Seq wird verwendet, um die Bereichsinformation für Fehler in F_s zu aktualisieren.

Wenn alle Fehler in F_T erkannt sind (**Fig.** 14, Zeilen 11–15), dann wird unter Verwendung einer anderen Prozedur als das standardbinäre Suchverfahren die neue Sequenz gewählt. Dann wird die Menge F_c (**Fig.** 14, Zeile 11) berechnet. Diese Menge enthält alle Fehler mit Detektionszeiten, die Vektoren in der Sequenz Seq entsprechen, aber es ist bekannt, daß diese Fehler von der Sequenz Seq nicht erkannt werden können. Die Vektoren, die zu restaurierten Sequenzen für Fehler in F_c gehören, müssen erneut ausgeschlossen werden. Zeile 12 in **Fig.** 14 ermittelt den Index start des ersten Vektors, der nicht zu restaurierten Sequenzen für irgendeinen Fehler in F_c gehört. Die nächste Sequenz, die auf Zeile 6 gewählt wird, ist $\{v_{start}, \ldots, v_{base}\}$ ResSeq. Das ist eine deutlich kürzere Sequenz.

Wenn schließlich ein Fehler in F_T nicht erkannt wird, gibt es eine Möglichkeit neue Ziele zu finden. UPDATE TARGETS ist eine erfindungsgemäße Ausführung des Verfahrens, das neue Ziele findet. Da neue Ziele in F_T eingebracht werden können, werden die Grenzen low und high (**Fig.** 14, Zeilen 18–19) neu berechnet. Wichtiger ist, daß die nächste Sequenz unter Verwendung des binären Suchverfahrens gewählt wird. Eine Sequenz größer als Seq wird in der nächsten Iteration berücksichtigt werden müssen.

Die erfindungsgemäße überlappende Validation und Verfeinerung verbessert deutlich die Durchführung bei der Restauration eines Segments. Wie nachstehend dargelegt wird, beschleunigen diese Verfahren die Kompaktierungsverfahren um einen Faktor von 2 für ein großes Indstriedesign.

BEISPIEL

Anhand von **Fig.** 15 wird ein Beispiel der beschleunigten Version des erfindungsgemäßen Verfahrens erläutert. Die Testmenge besteht aus dreißig Vektoren $v_1, \ldots, v_{30}$ und die Fehlerliste hat acht Fehler $F_u = \{f_1, \ldots, f_8\}$. Das Beispiel zeigt die Detektionszeit und alle Teilsequenzen, die einen Fehler unter der Annahme eines unbekannten ursprünglichen Zustandes erkennen. Beispielsweise Fehler f_3 wird erkannt, nach der Anwendung des Vektors v_{19}. Daher ist $D[f_3] = 19$.

Nur die Teilsequenz $v_{16}, \ldots, v_{19}$ erkennt diesen Fehler beim Start ausgehend von einem unbekannten ursprünglichen Zustand. Jeder Fehler wird von genau einer Teilsequenz erkannt, mit der Ausnahme von Fehler f_3, wie in **Fig.** 15 dargestellt ist. Dieser Fehler wird von zwei Teilsequenzen $v_1, \ldots, v_{22}$ und $v_{22}, \ldots, v_{28}$ erkannt.

Für dieses Beispiel benötigt das beschleunigte Verfahren zwei Verfeinerungsphasen. Zwei Segmente werden gefunden: $\{v_3, \ldots, v_{14}\}$ und $\{v_{17}, \ldots, v_{30}\}$. Nachstehend wird das Verfahren beschrieben, das für das Auffinden des ersten Segments verwendet wird. Das zweite Segment wird auf ähnliche Weise gefunden.

Das erste Segment $v_{17}, \ldots, v_{30}$ wird wie folgt hergeleitet. Zuerst wird das 2-ϕ*-Verfahren von **Fig.** 7 berücksichtigt. Da dies die erste Iteration ist, gibt es keine Vektoren in ResSeq. Zeilen 2–4 sind Initialisierungsschritte, die benötigt werden, bevor ein neues Segment hergeleitet wird. Die Variable base = 30, da die höchste Detektionszeit für den Fehler f_5 ist und $D[f_8] = 30$ ist. Die Zielfehlerliste F_T hat nur einen Fehler f_8. Die Prozedur RESET_LOW_HIGH initialisiert die Arrays LOW und HIGH. Für irgendein f aus F_u gilt LOW[f] = 1 und HIGH[f] = 31.

Das Verfahren OVERLAPPED_VALIDATION: Ursprünglich ist $F_T = f_8$. Die Variable start = 31, da $HIGH[f_8] = 31$. Das Restaurationsverfahren berücksichtigt sieben Sequenzen in der nachstehenden Reihenfolge.

1. Die erste für die Fehlersimulation berücksichtigte Sequenz ist v_{30}, da in Zeile 4 low = 30 ist. Die Fehlermenge $F_s = \{f_8\}$, da alle anderen Fehler in F_T Detektionszeiten haben, die kleiner als 30 sind. Eine falsche Simulation eines Vektors v_{30} detektiert keine Fehler in F_s. Die Bereichsinformation wird unter Verwendung der Prozedur UPDATE LOW HIGH wie folgt aktualisiert:
$HIGH[f_8] = 30$. Die Prozedur UPDATE TARGETS erzeugt keine neuen Ziele. Der erste Fehler, der in diesem Verfahren berücksichtigt wird, ist $f_p = f_7$ (**Fig.** 11, Zeile 3). Die Variable high = $HIGH[f_8] = 30$. Da $D[f_p] = 29$, ist high > $D[f_p]$ und Fehler f_7 ist kein neues Ziel.
2. Die nächste für die Fehlersimulation berücksichtigte Sequenz ist Sep = $\{v_{29}, v_{30}\}$, da in **Fig.** 9, Zeile 4 low = 29 ist. Die Menge F_T enthält immer noch nur einen Fehler f_8. Da die Detektionszeiten von f_7 und von f_8 größer oder gleich als 29 sind, ist jedoch $F_s = \{f_7, f_8\}$. Eine Fehlersimulation von Seq erkennt auch keinen neuen Fehler F_T. Die Bereichsinformation wird wie folgt aktualisiert: $HIGH[f_7] = HIGH[f_8] = 29$. Die Prozedur UPDATE_TARGETS stellt ein neues Ziel wie folgt bereit. Der erste in diesem Verfahren berücksichtigte Fehler ist $f_p = f_7$. Die Variable high = $HIGH[f_8] = 29$. Da $D[f_7] = 29$ ist, ist high $\leq D[f_7] + 1$. Außerdem ist $HIGH[f_7] \leq D[f_7]$. Daher ist f_7 ein neues Ziel und wird in die Zielfehlerliste F_T eingefügt. Die Menge F_{det} hat keine Fehler. Der zweite berücksichtigte Fehler ist $f_p = f_6$. Die Variable high = Minimum $HIGH[f_7]$, $HIGH[f_8] = 29$. Da $D[f_6] = 26$ ist, ist high > $D[f_6] + 1$ und f_6 ist kein neues Ziel.
3. Die nächste berücksichtigte Sequenz ist Seq = $\{v_{28}, \ldots, v_{30}\}$. Für diesen Durchlauf ist $F_T = F_s = \{f_7, f_8\}$. Die Fehlersimulation von Seq erkennt wiederum keinen Fehler in F_s. Die Bereichsinformation wird wie folgt aktualisiert: $HIGH[f_7] = HIGH[f_8] = 28$. Aus dem Verfahren UPDATE_TARGETS resultiert kein neues Ziel.

4. Die nächste berücksichtige Sequenz ist Seq = $\{v_{26}, \ldots, v_{30}\}$. Die Menge $F_T = \{f_7, f_8\}$ und $F_s = \{f_6, f_7, f_8\}$. Der Fehler f_6 wird in F_s berücksichtigt, da $D[f_6] = 26 \geq 26$ und $HIGH[f_6] = 31 > 26$ ist. Die Fehlersimulation führt zu der Erkennung des Fehlers f_8. Die Bereichsinformation wird wie folgt aktualisiert:
$HIGH[f_6] = HIGH[f_7] = LOW[f_8] = 26$. Das Verfahren UPDATE_TARGETS (1) erzeugt ein neues Ziel f_6 und (2) der Fehler f_8 wird als erkannt gekennzeichnet, und aus F_T und F_u entfernt. Der Fehler f_6 ist ein neues Ziel, da die restaurierte Sequenz für f_6 mit der Sequenz $v_{26}, \ldots, v_{30}$ überlappen wird.
5. Die nächste berücksichtigte Sequenz ist Seq = $\{v_{22}, \ldots, v_{30}\}$. Die Menge $F_T = \{f_6, f_7\}$ und $F_s = \{f_5, f_6, f_7\}$. Fehler f_5 ist wiederum in F_s enthalten, da seine Detektionszeit 22 ist. Die Fehlersimulation von Seq führt zu der Erkennung von f_5. Die Bereichsinformation wird wie folgt aktualisiert:
$HIGH[f_6] = HIGH[f_7] = LOW[f_5] = 22$. Das Verfahren UPDATE_TARGETS (1) erzeugt keine neuen Ziele, aber (2) der Fehler f_5 wird als erkannt gekennzeichnet, und von F_T und F_u entfernt.
6. Die nächste berücksichtige Sequenz ist Seq = $\{v_{14}, \ldots, v_{30}\}$. Die Menge $F_T = \{f_6, f_7\}$ und $F_s = \{f_1, f_2, f_3, f_4, f_6, f_7\}$. Die Fehler $f_1, \ldots, f_4$ sind in F_s enthalten, da ihre Detektionszeiten größer oder gleich 14 sind. Die Fehlersimulation von Seq führt zu der Erkennung von allen Fehlern in F_s mit der Ausnahme von f_1. Die Bereichsinformation wird wie folgt aktualisiert:
$LOW[f_2] = LOW[f_3] = LOW[f_4] = LOW[f_6] = LOW[f_7] = HIGH[f_1] = 14$. Das Verfahren UPDATE_TARGETS erzeugt keine neuen Ziele. Da alle Ziele in F_T erkannt sind, wird die überlappte Validationsphase mit der Sequenz $v_{14}, \ldots, v_{30}$ beendet.

Das Verfahren SEGMENT_FOUND: Ursprünglich ist $F_T = \{f_6, f_7\}$ und low = 14. Daher erkennt die Sequenz $v_{14}, \ldots, v_{30}$ alle Fehler in F_T. Die Menge F_c wird wie folgt berechnet. Alle Fehler in $F_u = \{f_1, f_2, f_3, f_4, f_6, f_7\}$ haben Detektionszeiten größer oder gleich 14. Nur ein Fehler f_1 jedoch hat $HIGH[f_1] \leq 14$. Daher ist $F_c = \{f_1\}$. Die erste für die Fehlersimulation vorgeschlagene Sequenz ist Seq = $\{v_{15}, \ldots, v_{30}\}$. Die Menge $F_s = \{f_2, f_3, f_4, f_6, f_7\}$. Die Fehlersimulation zeigt, daß alle Fehler in F_s erkannt sind. Die Bereichsinformation wird wie folgt aktualisiert:
$LOW[f_2] = LOW[f_3] = LOW[f_4] = LOW[f_6] = LOW[f_7] = 15$. Da alle Fehler in F_T erkannt sind; kann ein Segment innerhalb des Segments Seq möglich sein. Alle Fehler in F_T werden als unerkannt gekennzeichnet, und die kürzere Sequenz $\{v_{15}, \ldots, v_{30}\}$ wird als Segment geprüft. Die Menge F_c wird erneut neu berechnet und dabei erkannt, daß F_c keine Fehler hat. Daher hat die Sequenz $\{v_{15}, \ldots, v_{30}\}$ ein Segment.
Das Verfahren OVERLAPPED_REFNEMENT: Diese Phase identifiziert das Segment $\{v_{17}, \ldots, v_{30}\}$. Zu Beginn der Verfeinerungsphase ist $F_T = \{f_6, f_7\}$. Aus den Zeilen 1 und 2 des Verfeinerungsverfahrens von **Fig.** 14 folgt, daß high = 22 und low = 15 ist. Daher ist der erste Vektor des Segments irgendwo zwischen den Vektoren v_{15} und v_{22}. Das Verfeinerungsverfahren berücksichtigt fünf Sequenzen in der nachfolgenden Reihenfolge:

1. Die erste berücksichtigte Sequenz ist Seq = $\{v_{18}, \ldots, v_{30}\}$ (**Fig.** 14, Zeile 6), da opf = [(low + high)/2] = 18 ist, wie bei dem binären Suchverfahren der Verfeinerungsphase des grundlegenden Verfahrens ermittelt wird. Die Mengen $F_T = \{f_6, f_7\}$ und $F_s = \{f_2, f_3, f_4, f_5, f_7\}$. Die Fehlersimulation von Seq zeigt, daß die Fehler f_4, f_6 und f_7 erkannt sind. Die Bereichsinformation wird wie folgt aktualisiert: $LOW[f_4] = LOW[f_6] = LOW[f_7] = HIGH[f_2] = HIGH[f_3] = 18$. Da alle Fehler in F_T erkannt sind, wird die Menge F_c berechnet und $F_c = \{f_2, f_3\}$. Da F_c Fehler hat, wird daher die nächste Sequenz nicht unter Verwendung des binären Standard-Suchverfahrens gewählt. Die Variable start = 1 + 19 = 20 und die Variable low = 18.
2. Die nächste berücksichtigte Sequenz ist Seq = $\{v_{20}, \ldots, v_{30}\}$, da in **Fig.** 14, Zeile 5, opt = start = 20 ist. Zu dieser Zeit wird F_T nicht verändert und $F_s = \{f_4, f_6, f_7\}$. Die Fehlersimulation von Seq zeigt, daß die Fehler f_6 und f_7 erkannt werden, aber f_4 nicht erkannt wird. Die Bereichsinformation wird wie folgt aktualisiert: $LOW[f_6] = LOW[f_7] = HIGH[f_4] = 20$. Da alle Fehler in F_T erkannt werden, wird die Menge F_c berechnet und $F_c = \{f_4\}$. Da F_c Fehler hat, wird die nächste Sequenz nicht unter Verwendung des binären Standardsuchverfahrens gewählt. Die Variable start = 1 + 21 = 22 und die Variable low = 20.
3. Die nächste berücksichtigte Sequenz ist Seq = $\{v_{22}, \ldots, v_{30}\}$ (**Fig.** 14, Zeile 6) Wiederum bleibt $F_T = \{f_6, f_7\}$ unverändert. Jedoch ist $F_s = \{\}$, da Seq während der OVERLAPPED_VALIDATION-Phase bereits bei einer Fehlersimulation berücksichtigt wurde. Daher ist keine Fehlersimulation nötig. Die Fehler f_6 und f_7 können von Seq nicht erkannt werden. Die Bereichsinformation bleibt unverändert. $HIGH[f_6] = HIGH[f_7] = 22$. Nun werden unter Verwendung des Verfahrens UPDATE_TARGETS neue Ziele identifiziert.

Als nächstes wird das Verfahren UPDATE_TARGETS von **Fig.** 11 ausgeführt. Ursprünglich ist $F_T = \{f_6, f_7\}$ und $F_u = \{f_1, f_2, f_3, f_4, f_6, f_7\}$.

(a) Der erste berücksichtigte Fehler ist $f_p = f_4$ (**Fig.** 11, Zeile 3). Bei der Berücksichtigung von allen Fehlern in F_T wird ermittelt, daß high = 22 ist (**Fig.** 11, Zeile 4). Da high $\leq D[f_4] + 1$ ist und $HIGH[f_4] = 20 \leq D[f_4]$ ist, gibt es ein neues Ziel f_4, das in F_T aufgenommen wird. Die Menge F_{det} wird berechnet und $F_{det} = 0$ (Zeile 10), da $LOW[f_6]$ und $LOW[f_7] = 20$ ist und high = 22.
(b) Der zweite berücksichtigte Fehler ist $f_p = f_3$. Da $D[f_3] = 19$ ist und high = 20 ist, ist Fehler f_3 ein neues Ziel. Erneut wird f_3 zu F_T hinzugefügt. Die Zielfehlerliste wird aktualisiert auf $F_T = \{f_3, f_4\}$. Die Menge F_{det} wird berechnet und $F_{det} = \{f_6, f_7\}$, da $LOW[f_6] = LOW[f_7] = 20 \geq$ high. Diese beiden Fehler werden als erkannt gekennzeichnet. Sie werden aus F_T und F_u entfernt. Jetzt ist $F_T = \{f_3, f_4\}$.
(c) Der dritte berücksichtigte Fehler ist $f_p = f_2$. Da $D[f_2] = 18$ ist und high = 18 ist, ist Fehler f_2 ein neues Ziel. Wiederum wird f_2 zu F_T hinzugefügt. Die Zielfehlerliste wird aktualisiert auf $F_T = \{f_2, f_3\}$. Die Menge F_{det} wird bestimmt und $F_{det} = \{f_4\}$. Der Fehler f_4 wird als erkannt gekennzeichnet. Dieser Fehler wird aus F_T und F_u entfernt. Nun ist $F_T = \{f_2, f_3\}$.
(d) Der letzte berücksichtigte Fehler ist $f_p = f_1$. Da $D[f_1] = 14$ ist und high = 18 ist, ist Fehler f_1 kein neues Ziel.

Keine neuen Ziele sind möglich.

4. Daraufhin kehrt die Ausführung auf Zeile 18 des Verfahrens OVERLAPPED_REFINEMENT (**Fig.** 14) zurück. An diesem Punkt ist $F_T = \{f_2, f_3\}$. Da sich F_T verändert hat, werden die Grenzen neu berechnet und high = 18 und low = 15. Man beachte, daß die nächste Sequenz unter Verwendung des binären Suchverfahrens ausgewählt wird. Die nächste berücksichtigte Sequenz ist $Seq = \{v_{16}, \ldots, v_{30}\}$, da aus **Fig.** 14, Zeile 5 folgt, daß opt = 16 ist. Die Menge F_s ist die gleiche wie F_T. Die Fehlersimulation von Seq zeigt, daß beide Fehler f_2 und f_3 erkannt sind. Die Bereichsinformation wird wie folgt aktualisiert $LOW[f_2] = LOW[f_3] = 16$. Da alle Fehler in F_T erkannt sind, ist es möglich, daß das Segment für F_T kürzer als Seq sein kann. Die nächste Sequenz wird nicht unter Verwendung des binären Suchverfahrens ausgewählt. Die Variable statt = 1 + 16 = 17, und die Variable low = 16.
5. Die letzte berücksichtigte Sequenz ist $Seq = \{v_{17}, v_{30}\}$, da aus Zeile 16 von **Fig.** 14 hervorgeht, daß opt = 17 ist. Jetzt ist $F_s = F_T = \{f_2, f_3\}$. Die Fehlersimulation von Seq zeigt, daß f_2 erkannt wird, aber f_3 nicht erkannt wird. Die Bereichsinformation wird wie folgt aktualisiert: $LOW[f_2] = HIGH[f_3] = 17$. Es gibt eine Gelegenheit ein neues Ziel zu finden. Das Verfahren UPDATE_TARGETS (1) findet kein neues Ziel, aber (2) es ermittelt, daß f_2 erkannt wird. Dieser Fehler wird aus F_s und F_u entfernt. Die Werte von high und low werden neu berechnet und high = 17 und low = 16. Das verletzt die Bedingung der while-Schleife (**Fig.** 14, Zeile 4) und das Verfeinerungsverfahren wird beendet.

Versuchsergebnisse

Die vorstehend beschriebenen erfindungsgemäßen Vektorrestaurationsverfahren wurden als Teil eines statischen Fest-Sequenz-Kompaktierungssystems implementiert. Die Implementation wurde mit dem vorstehend beschriebenen grundlegenden Verfahren und dem vorteilhaften Verfahren durchgeführt. Um einen geeigneten Vergleich zu ermöglichen, wurde das lineare Vektorrestaurationsverfahren ebenfalls implementiert. Die Standardnäherung führt die Simulation eines einzigen Fehlers während des Vektorrestaurationsverfahrens durch. I. Pomeranz und S. M. Reddy "Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits" Proceedings Int. Conf. on Computer Design, S. 360–365, 1997, University of Iowa, August 1997. Diese Annäherung wurde erweitert, um eine Vielzahl von Fehlern während der Vektorrestauration zu berücksichtigen. Bei dieser Implementation werden Fehler mit gleichen Detektionszeiten simultan für die Restauration berücksichtigt. Daher hat das Verfahren den Vorteil eines parallelen Fehlersimulators, der in der Implementation verwendet wird. Diese Implementation des linearen Vektorrestaurationsverfahrens wird als LVR* bezeichnet.

Für ISCAS Wertungsschaltkreise und einige industrielle Designs sind experimentelle Resultate bekannt aus F. Brglez, D. Bryan und K. Kozminski "Combinational profiles of Sequential benchmark circuits", Int. Symposium on Circuits and Systems, S. 1929–1934, Mai 1989. Die neuen Verfahren werden mit der LVR*-Annäherung verglichen. Tabelle 1 von **Fig.** 16 zeigt die Resultate für ISCAS-Schaltkreise. Tabellen 2 von **Fig.** 17 zeigt Resultate von verschiedenen industriellen Designs. Alle Tabelle zeigen eine Anzahl von Vektoren in der ursprünglichen und kompaktierten Vektormenge. Die Kompaktierungsqualität wird in Prozent der Reduktion des ursprünglichen Vektortests angegeben. Die CPU-Sekunden sind für eine Sun UltraSPARC work station angegeben. CPU-Sekunden werden plattformspezifisch angegeben. Um eine Vorstellung von der Komplexität des Verfahrens zu geben, ist auch die Zeit für eine falsche Simulation von allen Fehlern unter Verwendung der ursprünglichen Vektormenge angegeben. Diese Zeiten werden angegeben, unter der Spalte Initial Sim. time. Die Spalten LVR*, 2-ϕ und 2-ϕ* geben die CPU-Sekunden an, die jeweils für die Kompaktierung unter Verwendung der linearen Vektorrestauration, der Zweiphasenrestauration und der beschleunigten Zweiphasenrestauration benötigt werden. Vektorrestauration auf der Grundlage von Kompaktierungsannäherungen behalten den Fehlerumfang (fault coverage) der ursprünglichen Vektormenge bei. Daher sind Fehlerumfangszahlen nicht angegeben.

Die ursprüngliche Fehlermenge, die für ISCAS-Schaltkreise verwendet wurde, wurde unter Verwendung des Testet-Generators HITEC erhalten, der beschrieben ist in T. M. Niermann und J. H. Patel, "HITEC: A test generation package for Sequential circuits" Proc. European Conf. Design Automation (EDAC), S. 214–218, März 1991. Die Resultate für die Kompaktierung auf ISCAS-Schaltkreisen sind in Tabelle 1 angegeben. Die Kompaktierung mit den vorgeschlagenen erfindungsgemäßen Restaurationsverfahren ist für große ISCAS-Designs bedeutend schneller. Für ISCAS-Designs ist die Kompaktierung unter Verwendung der beschriebenen Verfahren ungefähr zweimal schneller für eine falsche Simulation einer ursprünglichen Vektormenge.

Die Anzahl der GATES und Anzahl der FLIP FLOPS in dem betrachteten industriellen Design der Experimente ist in Tabelle 2 angegeben. Industrielle Designs haben einige nicht Bool'sche Primitive, sowie Tristatpuffer, bidirektionale Puffer und Busse. Sie haben außerdem Setz/Rücksetz FLIP FLOPS und Vielfachtakte.

Ursprüngliche Testmengen für diese Schaltkreise wurden mit einem auf dem Markt erhältlichen Testgenerator erzeugt.

Auch Tabelle 2 zeigt Resultate, die unter Verwendung des erfindungsgemäßen vorstehend beschriebenen Beschleunigungsverfahrens erzielt wurden. Resultate, die unter Verwendung des grundlegenden Verfahrens von **Fig.** 4 erzielt wurden, sind in Spalte 24 angezeigt. Resultate, die unter Verwendung der Beschleunigungsverfahren erzielt wurden, sind in Spalte 2-ϕ* angezeigt. Aus der Tabelle geht hervor, daß die Kompaktierungszeit ungefähr das 2- bis 10-fache der ursprünglichen Fehlersimulationszeit beträgt. Die Kompaktierung mit der beschleunigten Zweiphasenrestauration läuft 2- bis 5-fach schneller als die lineare Vektorrestauration. Beispielsweise will für das industrielle Design p29 mit der beschleunigten Zweiphasenrestauration die Kompaktierung in 12380 s fertiggestellt werden. Für den gleichen Schaltkreis wurde für die Kompaktierung mit der verbesserten linearen Vektorrestauration 41616 s benötigt. Unter Verwendung der vorstehend beschriebenen erfindungsgemäßen Restaurationsverfahren war das Kompaktierungssystem in der Lage, eine Kompaktierung auf großen Designs mit ungefähr 200.000 Gates und 5000 Flip Flops fertigzustellen, während mit der LVR* selbst nach 2 CPU-Tagen eine Kompaktierung nicht fertigstellen konnte. Die überlappte Validation und die Verfeinerungsverfahren stellten sich als überaus nützlich für große Designs heraus. Die Resultate für diese Verfahren sind in

Spalte 2-ϕ* angezeigt. Die 2-ϕ*-Version ist besonders wirkungsvoll für Schaltkreise mit großen Segmenten. In dem industriellen Design p306 gibt es beispielsweise ein Segment mit 2666 Vektoren und die Verwendung des 2-ϕ*-Verfahrens führt zu einer Verbesserung der CPU-Sekunden um mehr als einen Faktor 2.

Die experimentellen Resultate zeigen die Überlegenheit des neuen Vektorrestaurationsverfahrens. Die erfindungsgemäßen Verfahren verbessern deutlich die Laufzeiten einer Vektorrestauration auf der Grundlage von statischen Kompaktierungsverfahren.

RESUMÉ

Die vorstehend beschriebenen erfindungsgemäßen Vektorrestaurationsverfahren sind den herkömmlichen Verfahren deutlich überlegen. Die neuen Verfahren führen eine Vektorrestauration durch, indem schrittweise zwei Phasen wiederholt werden:
Validation und Verfeinerung. Die erfindungsgemäßen Verfahren können Teilsequenzen restaurieren, die zusätzlichen Bedingungen entsprechen, wie: (1) Zielfehler werden unter der Annahme eines bekannten ursprünglichen Zustands vor der Anwendung der Teilsequenz erkannt, oder (2) eine restaurierte Teilsequenz bildet ein Segment. Teilsequenzen, die zusätzliche Eigenschaften haben, können in statischen Testsequenzkompaktierungsverfahren und Fehlerdiagnoseverfahren verwendet werden. Die vorliegende Erfindung betrifft außerdem Beschleunigungsverfahren für eine Vektorrestauration. Resultate aus Experimenten auf ISCAS-Designs und einigen großen industriellen Designs bestätigen die Anwendbarkeit der vorliegenden Verfahren.

Abwandlungen und Variationen der vorliegenden Erfindung sind für den Fachmann aufgrund der vorstehenden technischen Lehre selbstverständlich. Obwohl daher nur spezielle Ausführungen der Erfindung detailliert beschrieben wurden, ist klar, daß die vorliegende Erfindung vielfach abgewandelt werden kann, ohne den Rahmen der Erfindung zu verlassen.

Patentansprüche

1. Verfahren zur Restauration einer Sequenz von Testvektoren zum Testen eines Systems, wobei das System eine Fehlermenge hat, die von einer Sequenz von Testvektoren erkennbar ist, und wobei eine Teilmenge der Fehlermenge Zielfehler genannt ist, und wobei das Verfahren eine Validationsphase und eine Restaurationsphase hat, und wobei die Validationsphase eine erste Teilsequenz der Testvektoren identifiziert, die die Zielfehler erkennen, und eine zweite Teilsequenz der Testvektoren identifiziert, die die Zielfehler nicht erkennen, und wobei die Restaurationsphase die kürzeste Teilsequenz zwischen der ersten Teilsequenz und der zweiten Teilsequenz identifiziert, die die Zielfehler erkennt.

2. Restaurationsverfahren für eine Sequenz von Testvektoren mit:
   (a) Zuordnung von Fehlern zu einer Fehlerliste;
   (b) Identifizierung einer Detektionszeit für jeden der Fehler;
   (c) Initialisierung einer Restaurationssequenzliste zu Null (nil);
   (d) Zuordnung von Fehlern aus der Fehlerliste mit hohen Detektionszeiten zu einer Zielfehlerliste;
   (e) Zuweisung von base gleich dem Minimum der höchsten Detektionszeiten, und eine Zeit, die einem ersten Vektor in der Restaurationssequenzliste entspricht;
   (f) Durchführung einer Validationsphase, die eine low Teilsequenz der Testvektoren identifiziert, die alle Fehler in der Zielfehlerliste erkennt, und eine high Teilsequenz identifiziert, die keinen Fehler in der Zielfehlerliste erkennt;
   (g) Durchführung einer Verfeinerungsphase, die eine kürzeste Teilsequenz zwischen der low Teilsequenz und der high Teilsequenz identifiziert, wobei die kürzeste Teilsequenz alle Fehler in der Zielfehlerliste erkennt;
   (h) Entfernung der Fehler aus der Fehlerliste, die auch in der Zielfehlerüste sind;
   (i) Aktualisierung der Restaurationssequenzliste, so daß sie die Vereinigung der Restaurationssequenzliste und der kürzesten Teilsequenz ist, die in Schritt (g) identifiziert ist; und
   (j) Wiederholung der Schritte (d) bis (i), bis die Fehlerliste leer ist.

3. Verfahren nach Anspruch 2, wobei in der Validationsphase zusätzliche Vektoren kontinuierlich zu der Restaurations-Sequenz addiert werden, und eine Fehlersimulation durchgeführt wird, bis alle Fehler in der Zielfehlerliste erkannt sind.

4. Verfahren nach Anspruch 2, wobei die Validationsphase von Schritt (f) die nachstehenden Schritte umfaßt:
   (f)(1) Zuordnung von low = base;
   (f)(2) Erstellung einer Sequenzliste, die eine Vereinigung einer Vektor-Sequenz low zu base und der Restaurationssequenz ist;
   (f)(3) Durchführung einer Simulation, um Fehler in der Zielfehlerliste unter Verwendung der Sequenzliste von Schritt (f)(2) zu simulieren;
   (f)(4) Setzen der Werte high = low, low = base - K, wobei K ein vorbestimmter Wert ist; und
   (f)(5) Wiederholung der Schritte (f)(2) bis (f)(4), bis alle Fehler in der Zielfehler-liste bekannt sind.

5. Verfahren nach Anspruch 2, wobei die Verfeinerungsphase mit der Restaurationssequenz durchgeführt wird, indem eine binäre Suche durchgeführt wird, um eine kürzeste Teilsequenz zu erkennen, die alle Fehler in der Zielfehlerliste erkennt.

6. Verfahren nach Anspruch 2, wobei die Verfeinerungsphase von Schritt (g) die folgenden Schritte umfaßt:
   (g)(1) Zuordnung von opt gleich einem Mittelpunkt zwischen high und low;
   (g)(2) Erstellung einer Sequenzliste, die eine Vereinigung einer VektorSequenz von opt bis base und der Restaurationssequenz ist;
   (g)(3) Durchführung einer Simulation von Fehlern der Zielfehler unter Verwendung der Sequenzliste von

Schritt (g)(2);

(g)(4) Wenn alle Fehler der Zielfehlerliste erkannt sind, wird low = opt zugewiesen, und wenn alle Fehler der Zielfehler nicht erkannt werden, wird high = opt zugewiesen; und

(g)(5) Wiederholung der Schritte (g)(1) bis (g)(4) solange low < high = 1 ist.

7. Verfahren nach Anspruch 4, wobei $K = x^i$, $x >= 2$ und $i = 0$ in der ersten Wiederholung ist und i wird in Wiederholungen um einen konstanten Wert erhöht, die von der ersten Wiederholung verschieden sind.

8. Beschleunigtes Restaurationsverfahren einer Sequenz von Testvektoren, das die nachstehenden Schritte umfaßt

(a) Identifikation von Testvektoren, eine Fehlerliste mit Fehlern, die unter Verwendung dieser Testvektoren erkannt werden können, und Detektionszeiten für die Fehler;

(b) Selektion von Fehlern, die einer Zielfehlerliste zugeordnet werden, falls die Fehler existieren;

(d) Durchführung einer überlappten Validation, so daß, wenn zwei Fehler in der Zielfehlerliste Restaurationssequenzen haben, die überlappen, dann werden die beiden Fehler vermischt und bilden einen Zielfehler,

(e) Durchführung einer überlappten Verfeinerung, wenn ein Segment, das den einen Zielfehler erkennt, in Schritt (d) existiert; und

(f) Wiederholung der Schritte (b) bis (e) solange in Schritt (b) Zielfehler existieren.

9. Verfahren nach Anspruch 8, wobei während der Durchführung der überlappten Validation das Restaurationsverfahren zweier Vektoren überlappt wird, falls restaurierte Sequenzen von zwei Fehlern gemeinsame Vektoren haben.

10. Verfahren nach Anspruch 8, wobei neue Zielfehler, die nicht in der Zielfehlerliste sind, zu der Zielfehlerliste hinzugefügt werden, solange wenigstens ein Fehler der Zielfehler während dem überlappten Validationsschritt unerkannt ist.

11. Verfahren nach Anspruch 8, wobei während der Durchführung der überlappten Verfeinerung ein Segment zwischen einer low Sequenz identifiziert wird, die alle Fehler in der Zielfehlerliste identifiziert, und eine high Teilsequenz bei der wenigstens ein Fehler in der Zielfehlerliste nicht identifiziert wird.

12. Verfahren nach Anspruch 8, wobei solange wenigstens ein Fehler der Zielfehler während der überlappten Verfeinerung nicht erkannt wird, und falls neue Fehler identifiziert werden, die nicht in der Zielfehlerliste sind, dann werden die neuen Fehler zu der Zielfehlerliste hinzugefügt.

13. Verfahren nach Anspruch 8, wobei Schritt (d) die folgenden Schritte umfaßt:

(d)(1) Zuordnung von HIGH(f) und LOW(f) für jeden Fehler in der Fehlerliste, wobei eine Teilsequenz von HIGH(f) bis zu einem letzten Vektor einen Fehler f nicht erkennt, und eine Teilsequenz von LOW(f) bis zu einem letzten Vektor den Fehler erkennt;

(d)(2) Setzen von low = 0;

(d)(3) Setzen von start gleich dem Minimum von HIGH(f) aller Fehler f;

(d)(4) Setzen von low gleich dem Maximum von start-L und low, wobei L ein vorbestimmter Wert ist;

(d)(6) Setzen einer Sequenz gleich der Vereinigung einer Teilsequenz von low bis base und der Restaurationssequenz;

(d)(7) Selektion von Fehlern für eine zweite Fehlerliste F^s, so daß $D[f] \geq$ low ist und HIGH[f] > low ist, für alle Fehler in der Fehlerliste;

(d)(8) Durchführung einer Simulation zur Überprüfung, ob Fehler in F^s von der Sequenz von Schritt (d)(6) erkannt werden;

(d)(9) Aktualisierung der Zielfehlerliste;

(d)(10) Aktualisierung von HIGH und LOW; und

(d)(11) Wiederholung der Schritte (d)(4) bis (d)(8), solange Fehler in der Zielfehlerliste nicht erkannt sind.

14. Verfahren nach Anspruch 8, wobei Schritt (e) die folgenden Schritte umfaßt:

(e)(1) Zuordnung von high gleich dem Minimum von HIGH[f] für alle Fehler f;

(e)(2) Zuordnung von low gleich dem Minimum von LOW[f] für alle Fehler f und einer Binärsuche gleich Yes;

(e)(3) Setzen von opt = (low + high)/2, falls Binary-Search Yes ist und opt = start, falls Binary-Search gleich No ist;

(e)(4) Erstellen einer Sequenzliste, die eine Vereinigung einer Vektorsequenz von opt bis base und der Restaurationssequenz ist;

(e)(5) Zuordnung von Fehlern zu einer zweiten Fehlerliste F^s, so daß $D[f] \geq$ opt HIGH[f] > opt für alle Fehler in der Fehlerliste ist.

(e)(6) Durchführung einer Simulation, um zu sehen, ob Fehler in F^s unter Verwendung der Sequenzliste erkannt werden;

(c)(7) Aktualisierung von low und high für Fehler in F^s;

(e)(8) Durchführung der folgenden Schritte, falls Fehler in der Zielfehlerliste erkannt werden:

(e)(8)(i) Berechnung einer Fehlerliste F_c, die alle Fehler enthält, die von der Sequenz nicht erkennbar sind und mit Detektionszeiten, die der Sequenzliste entsprechen;

(e)(8)(ii) Setzen von start = Maximum D[f] + 1 für alle Fehler in F_c;

(e)(8)(iii) Setzen von low gleich opt; und

(e)(8)(iv) Binary-Search = Yes, falls F_c gleich Null (nil) ist, und Binary-Search = No, falls F_c gleich Null (nil) ist;

(e)(9) Durchführung der folgenden Schritte, falls Fehler in der Zielfehlerliste nicht erkannt werden:

(e)(9)(i) Aktualisierung der Zielfehlerliste;

(e)(9)(ii) Setzen von high auf Maximum von HIGH[f] für alle Fehler in der Zielfehlerliste;

(e)(9)(iii) Setzen von low auf das Maximum von LOW[f] für alle Fehler in der Zielfehlerliste; und

(e)(9)(iv) Setzen von Binary-Search = Yes; und

(e) (10) Wiederholung von (e) (3) bis (e)(9) solange low < high - 1 ist.

15. Verfahren nach Anspruch 13, wobei Schritt (d)(10) die nachfolgenden Schritte umfaßt:

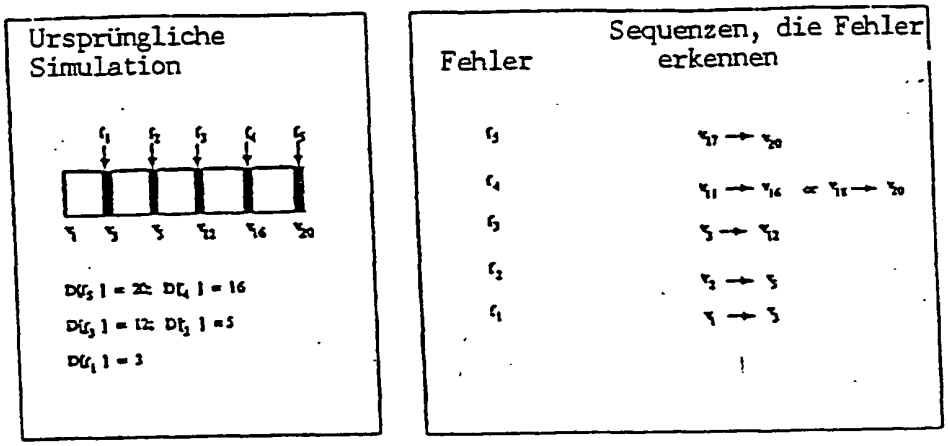
(d)(10)(i) Setzen von LOW[f] gleich dem Maximum von LOW[J] und low, falls ein Fehler f erkannt wird;
(d)(10)(ii) Setzen von HIGH(f) auf das Minimum von HIGH(f) und low, falls ein Fehler f nicht erkannt wird; und
(d)(10)(iii) Wiederholung der Schritte (d)(10)(i) bis (d)(10)(ii) für alle Fehler

16. Verfahren nach Anspruch 13, wobei Schritt (d)(9) die folgenden Schritte umfaßt:
(d)(9)(i) Selektion eines Fehlers f^p mit der höchsten D[f], die noch unerkannt ist, und die nicht in der Zielfehlerliste ist;
(d)(9)(ii) Setzen von high auf das Minimum HIGH[f] für alle Fehler in der Zielfehlerliste;
(d)(9)(iii) Hinzufügen von f^p zu der Zielfehlerliste, falls high $\leq D[f^p] + 1$ ist und $HIGH[f^p] \leq D[f^p]$ ist;
(d)(9)(iv) Selektion von Fehlern f aus der Zielfehlerliste mit LOW[f] high, und Entfernung der Fehler aus der Zielfehlerliste und aus der unerkannten Fehlerliste; und
(d)(9)(v) Wiederholung der Schritte (d)(9)(i) bis (d)(9)(iv), solange f_p zu der Zielfehlerliste hinzugefügt wird.

Hierzu 17 Seite(n) Zeichnungen

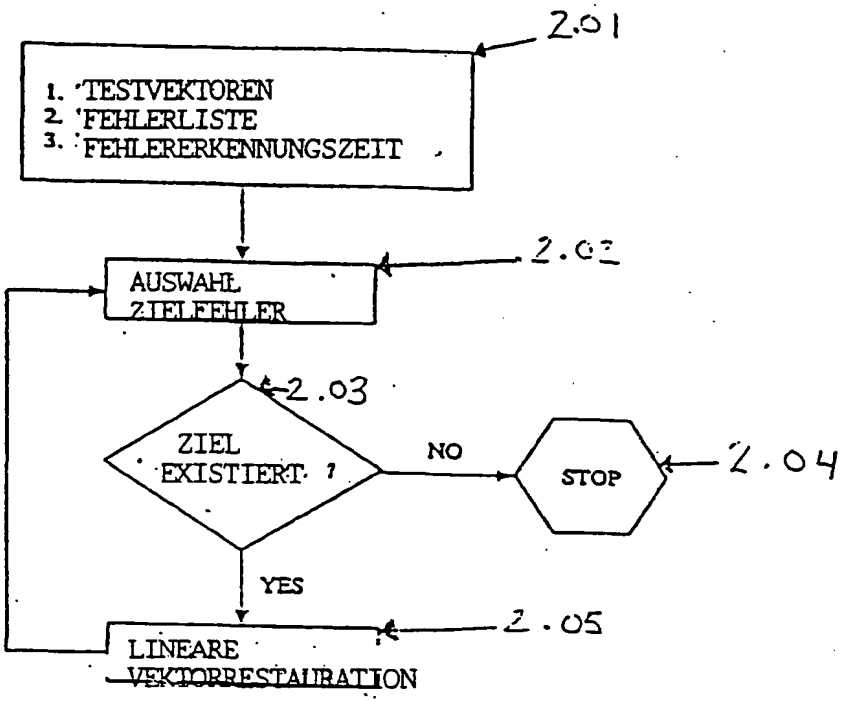
- Leerseite -

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



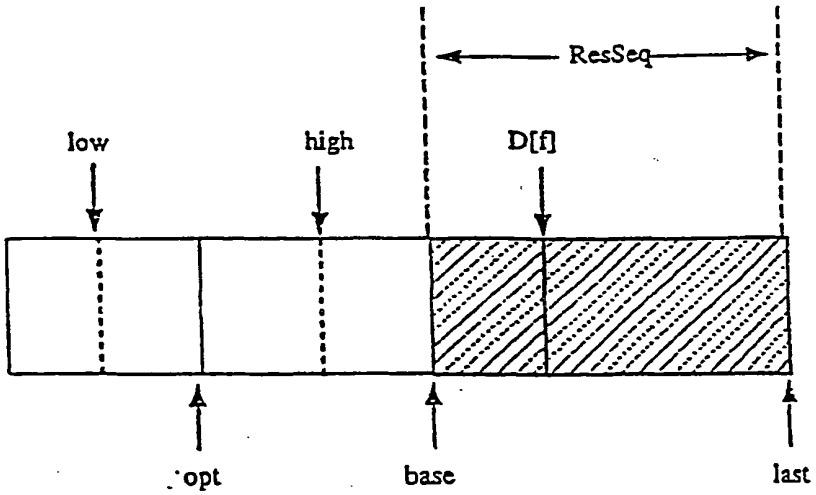
FIGUR 1

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



FIGUR 2

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



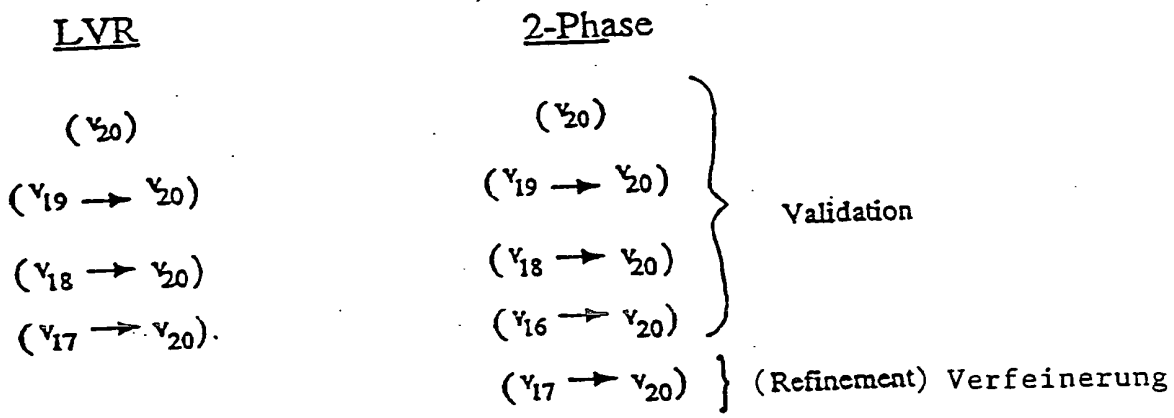
FIGUR 3

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

```
                    2φ(F_u, D)
    F_u :Input fault list    - Eingangsfehlerliste
    D : Detection times for all faults Detektionszeiten für alle
                                       Fehler
{
1.  ResSeq = Nil ;
2.  while (F_u ≠ Nil) {
3.      F_T = {f | f ∈ F_u & D[f] is highest} ;
4.      base = MINIMUM(D[f], FIRST(ResSeq));
        —— VALIDATION PHASE —— Validationsphase
5.      low = base ; i = 0;
6.      while (F_T not detected) {
7.        Seq = {v_low, ..., v_base} ∪ ResSeq ;
8.        FAULTSIMULATE (Seq, F_T) ;
9.        if (F_T not detected) {
10.         high = low ;
11.         low = base − 2^i ; i++;
12.       }
13.     }
14.     MARK_FAULTS_UNDETECTED (F_T) ;
        —— REFINEMENT PHASE —— Verfeinerungsphase
15.     while (low < high − 1) {
16.       opt = ⌊(low+high)/2⌋ ;
17.       Seq = {v_opt, ..., v_base} ∪ ResSeq ;
18.       FAULTSIMULATE (Seq, F_T) ;
19.       if (F_T is detected) {
20.         low = opt ;
21.         MARK_FAULTS_UNDETECTED(F_T) ;
22.       } else {
23.         high = opt ;
24.       }
25.     }
26.     MARK_FAULTS_DETECTED (F_T) ;
27.     F_u = F_u − F_T ;
28.     ResSeq = {v_low, ..., v_base} ∪ ResSeq ;
29. }
30. return (ResSeq) ;
}
```

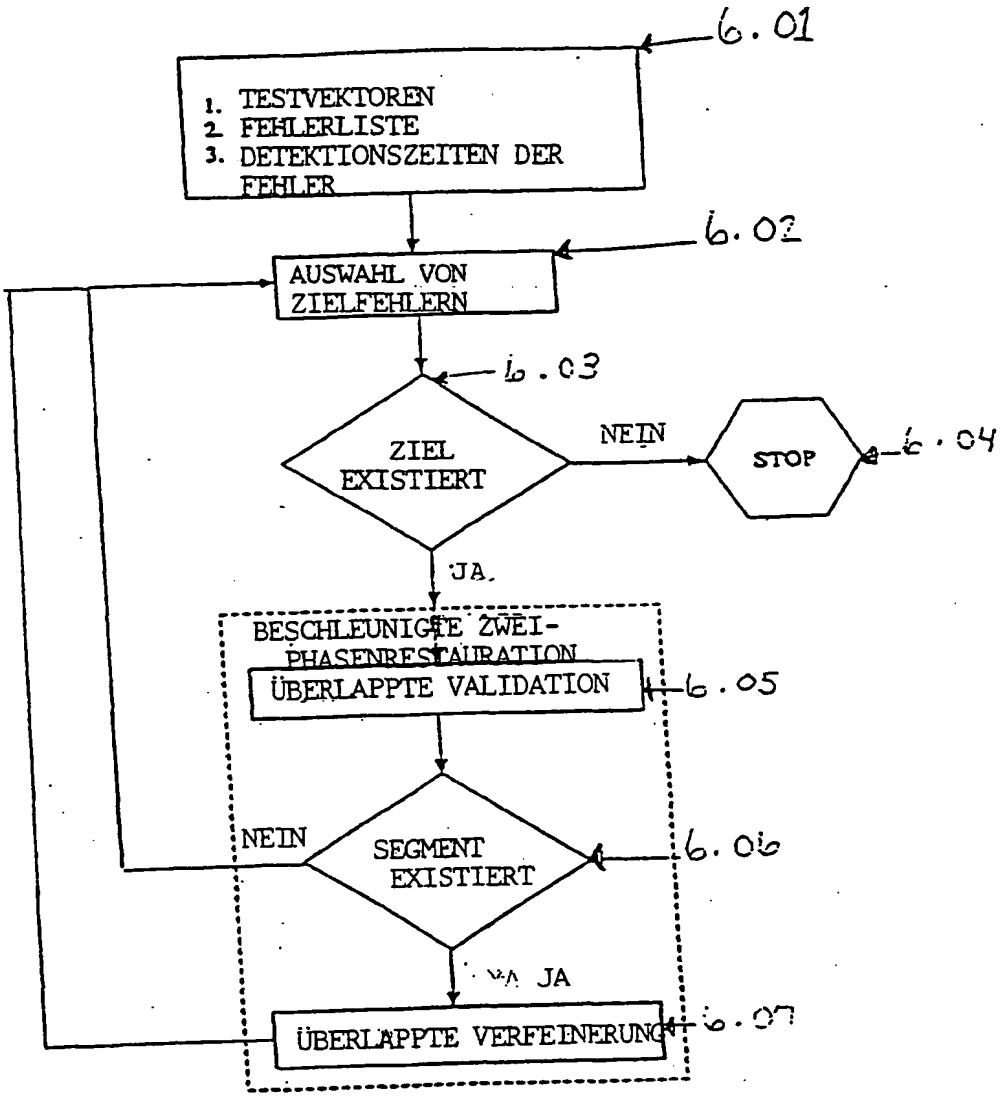
FIGUR 4

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



FIGUR 5

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



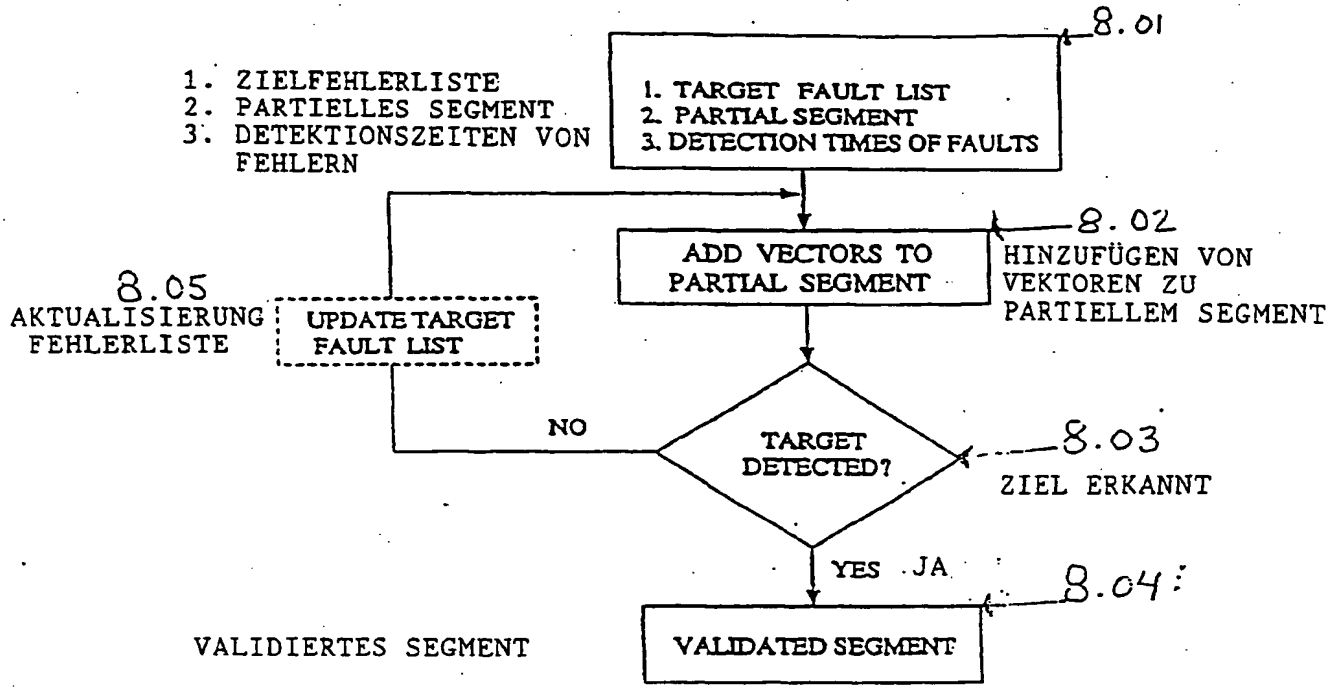
FIGUR 6

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

```
        2φ*(F_u, D)
    F_u : Input fault list  Eingangsfehlerliste
    D : Detection times for all faults  Detektionszeiten für alle
                                        Fehler
{
1.  ResSeq = Nil ;
2.  base = MAXIMUM_{f∈F_u}(D[f]);
3.  F_T = {f|f ∈ F_u&D[f] = base} ;
4.  RESET_LOW_HIGH (F_u, base+1);
5.  while (F_u ≠ Nil) {
6.      OVERLAPPED_VALIDATION (F_u, F_T, ResSeq, base);
7.      if (SEGMENT_FOUND(F_u, F_T, ResSeq, base)) {
8.        opt = OVERLAPPED_REFINMENT (F_u, F_T, ResSeq, base) ;
9.        ResSeq = {v_opt, ..., v_base} ∪ ResSeq ;
10.       base = MAXIMUM_{f∈F_u}(D[f]);
11.       F_T = {f|f ∈ F_u&D[f] = base} ;
12.       RESET_LOW_HIGH (F_u, base+1);
13.     }
14. }
15. return(ResSeq) ;
}
```

FIGUR 7

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



FIGUR 8

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

```
OVERLAPPED_VALIDATION(F_u, F_T, ResSeq, base)
F_u: Undetected fault list  UNERKANNTE FEHLERLISTE
F_T: Current target fault list  AKTUELLE ZIELLISTE
ResSeq: Set of Vectors restored for previous targets
base: Restoration for F_T begins from vector v_base
{
1.  i = low = 0;
2.  start = MINIMUM_{f∈F_T}(HIGH[f]);
3.  while (F_T not detected) {
4.      low = MAXIMUM(start − 2^i, low); i++;
5.      Seq = {v_low, ..., v_base} ∪ ResSeq;
6.      F_s = {f | f ∈ F_u & D[f] ≥ low & HIGH[f] > low};
7.      FAULT_SIMULATE (Seq, F_s);
8.      UPDATE_LOW_HIGH (F_s, low);
9.      UPDATE_TARGETS (F_T, F_u);
10. }
11. return;
}
```

ResSeq: Satz von Vektoren, die für frühere Ziele restoriert sind
base: Restoration für F_T beginnt von Vektor v_{base}

Figur 9

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

```
        UPDATE_LOW_HIGH(F, index)
    F: fault list being updated
    index: index of first vector in sequence used for fault simulation
{
1.  foreach (f ∈ F) {
2.      if (f is detected) {
3.          LOW[f] = MAXIMUM(LOW[f], index) ;
4.      } else {
5.          HIGH[f] = MINIMUM(HIGH[f], index) ;
6.      }
7.  }
8.  return;
}
```

F: aktualisierte Fehlerliste
Index: Index des ersten Vektors in der Sequenz, der für die Fehlersimulation verwendet wird

Figur 10

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

```
UPDATE_TARGETS(F_T, F_u)
F_T: current target fault list Aktuelle Zielfehlerliste
F_u: current set of undetected fault list Aktuelle Menge von
                                    unerkannten Fehlerliste
{
1.  do {
2.      Target_Changed = False ;
3.      f_p = (f | f ∈ F_u − F_T, D[f] is highest) ;
4.      high = MINIMUM_{f∈F_T}(HIGH[f]) ;
6.      if (high ≤ D[f_p] + 1 & HIGH[f_p] ≤ D[f_p]) {
7.        F_T = F_T ∪ f_p ;
8.        Target_Changed = True ;
9.      }
10.     F_det = {f | f ∈ F_T, LOW[f] ≥ high} ;
11.     F_T = F_T − F_det ;
12.     F_u = F_u − F_det ;
13.     MARK_FAULTS_DETECTED(F_det) ;
14. } while( Target_Changed == True)
15. return;
}
```

FIGUR 11

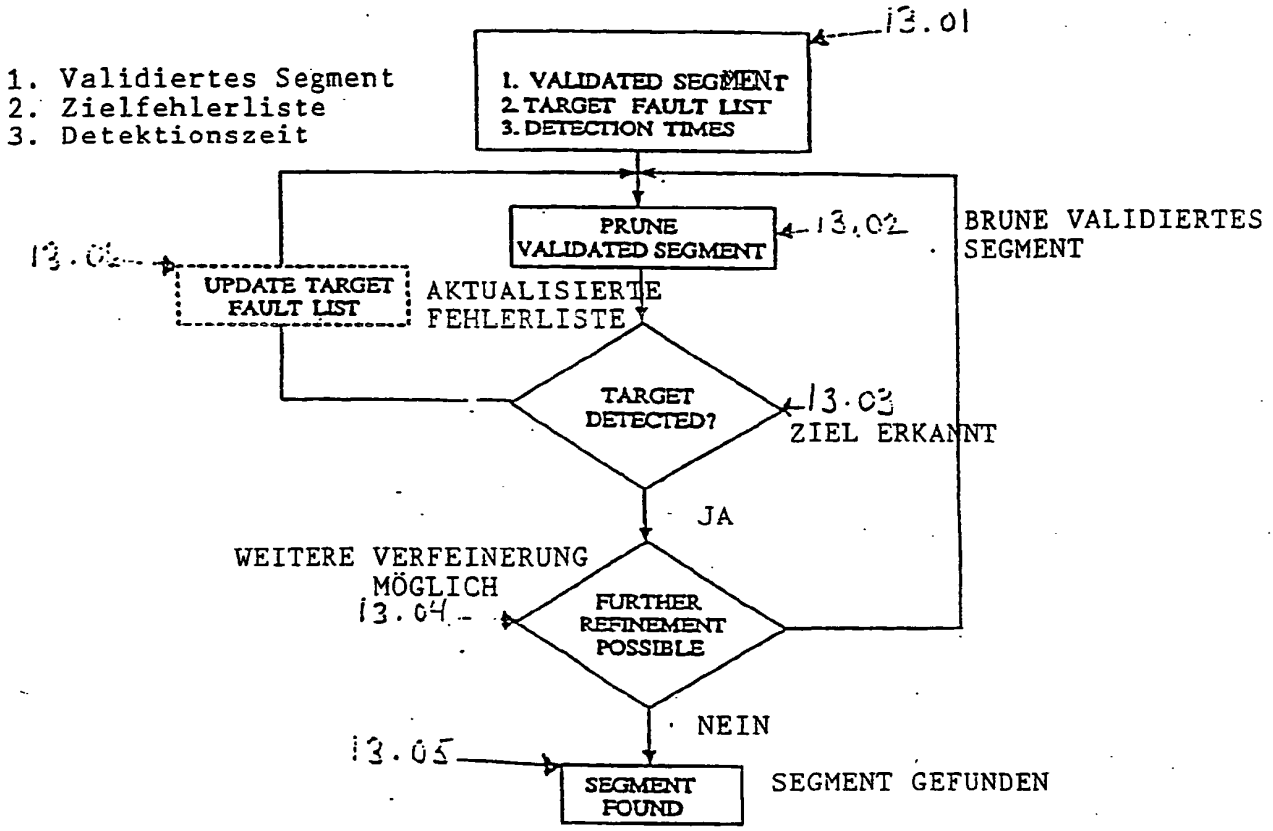
Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

```
SEGMENT_FOUND(F_u, F_T, ResSeq, base)
F_u: current undetected fault list aktuelle unerkannte Fehlerliste
F_T: current target fault list aktuelle Fehlerliste
ResSeq: sequence restored for previous targets
base: last vector of new segment (if any) is vector v_base
{
1.  status = Uncertain ;
2.  low = MINIMUM_{f∈F_T}(LOW[f]) ;
3.  while (status == Uncertain) {
4.      F_c = {f|(f ∈ F_u & D[f] ≥ low & HIGH[f] ≤ low};
5.      if (F_c == Nil) {
6.          status = Segment_found;
7.      } else {
8.          low = 1 + MAXIMUM_{f∈F_c}(D[f]) ;
9.          Seq = {v_low, ..., v_base} ∪ ResSeq ;
10.         F_s = {f| f ∈ F_u & D[f] ≥ low & HIGH[f] > low} ;
11.         FAULT_SIMULATE (Seq, F_s) ;
12.         UPDATE_LOW_HIGH (F_s, low) ;
13.         if (F_T is detected) {
14.             MARK_FAULTS_UNDETECTED (F_s) ;
15.         } else {
16.             UPDATE_TARGETS (F_T, F_u) ;
17.             status = No_segment ;
18.         }
19.     }
20. }
21. return(status) ;
}
```

ResSeq: restorierte Sequenz für frühere Ziele
base: LetzterVektor eines neuen Segments (falls vorhanden) ist Vektor v_{base}

FIGUR 12

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



FIGUR 13

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

```
OVERLAPPED_REFINEMENT(F_u, F_T, ResSeq, base)
F_u: current undetected fault list
F_T: current target fault list
ResSeq: restored sequence for previous targets
base: last vector of new segment is v_base
{
1.  high = MINIMUM_{f∈F_T}(HIGH[f]) ;
2.  low = MINIMUM_{f∈F_T}(LOW[f]) ;
3.  binary_search = Yes ;
4.  while (low < high − 1) {
5.      opt = (binary_search == Yes) ? (low+high)/2 : start ;
6.      Seq = {v_opt, ..., v_base} ∪ ResSeq ;
7.      F_s = {f | f ∈ F_u & D[f] ≥ opt & HIGH[f] > opt} ;
8.      FAULT_SIMULATE (Seq, F_s) ;
9.      UPDATE_LOW_HIGH (F_s, opt) ;
10.     if (F_T is detected) {
11.       F_c = {f | f ∈ F_u & D[f] ≥ opt & HIGH[f] ≤ opt} ;
12.       start = 1 + MAXIMUM_{f∈F_c}(D[f]) ;
13.       MARK_FAULTS_UNDETECTED(F_s) ;
14.       low = opt ;
15.       binary_search = (F_c == Nil) ? Yes : No ;
16.     } else {
17.       UPDATE_TARGETS (F_T, F_u) ;
18.       high = MINIMUM_{f∈F_T}(HIGH[f]) ;
19.       low = MINIMUM_{f∈F_T}(LOW[f]) ;
20.       binary_search = Yes ;
21.     }
22. }
23. return;
}
```

F_U: Aktuelle unentdeckte Fehlerliste

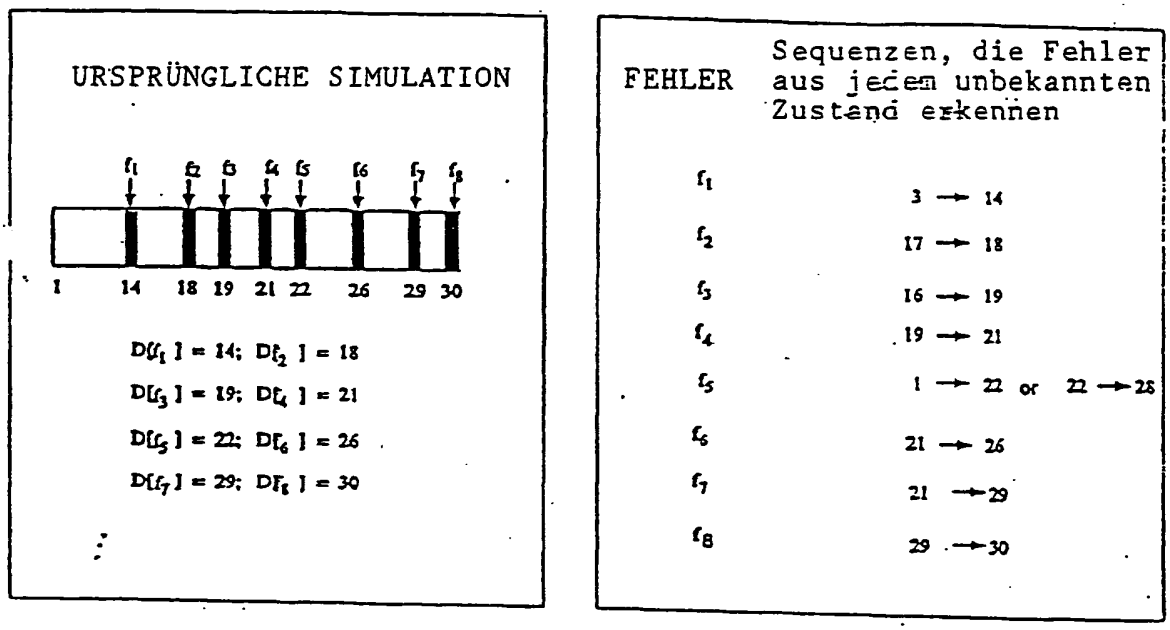
F_T: Aktuelle Zielfehlerliste

ResSeq: restorierte Sequenz für frühere Ziele

base: letzter Vektor eines neuen Segments ist v_{base}

FIGUR 14

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999



FIGUR 15

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

Tabelle 1: Kompaktierungsresultate für ISCAS-Schaltkreise mit HITEC-Testmengen

Ckt	VEKTOREN			ursprüngl. Sim.-Time	CPU Sekunden	
	Original	Compacted	% R		LVR*	2 ϕ*
s298	322	216	32	0.3 s	1.1 s	0.8 s
s344	127	61	51	0.2s	0.3 s	0.3 s
s382	2074	878	78	4.6s	52.4 s	6.0 s
s444	2240	1005	55	5.5s	16.0 s	7.5 s
s526	2258	1526	32	6.8s	82.5 s	12.0 s
s641	209	125	40	0.4s	0.7 s	0.8 s
s713	173	106	38	0.4s	0.7 s	0.8 s
s820	1115	790	29	2.1s	5.4 s	5.9 s
s832	1137	779	31	2.2s	6.9 s	6.5 s
s1196	435	281	35	1.1s	2.2 s	2.2 s
s1238	475	303	36	1.3s	2.5 s	2.7 s
s1423	150	134	10	2.2s	8.2 s	4.2 s
s1488	1170	828	29	4.4s	11.8 s	10 s
s1494	1245	855	31	4.7s	12.8 s	11 s
s5378	912	653	28	22.1s	91 s	39 s
s35932	496	202	59	198s	619 s	487 s

Vec: Testmengenlänge Time: Ausführungszeit
% R: Prozent Reduktion der ursprüngl. Testmengengröße

FIGUR 16

Nummer: DE 199 24 242 A1
Int. Cl.[6]: G 06 F 11/00
Offenlegungstag: 30. Dezember 1999

Tabelle 2: KOMPAKTIERUNGSRESULTATE FÜR DIE HERSTELLUNG VON SCHALTKREISEN

Ckt	Gates	FFs	VEKTOREN			ursprüngl.	CPU Sekunden		
			Orig.	Comp.	% R	Sim.Time:	LVR*	2-ϕ	2-ϕ*
p7A	7784	137	1702	989	41	98 s	606 s	199	185 s
p7B	6687	130	2267	1650	27	95 s	390 s	188	179 s
p12	8489	408	3510	2505	33	347 s	2019	635	589 s
p20	12145	487	2318	1894	18	301 s	4238	1966	877 s
p30	24824	995	1836	3192	37	186 s	2350	1203	974 s
p29	62617	935	9366	7201	23	2097 s	41616	13997	12380 s
p306	223962	5005	3499	3399	2	3520 s	NA	69829	31968 s

Gates: Anzahl von Gates
FFs: Anzahl von Flip-Flops
% R: Prozent Reduktion von ursprüngl. Testmengengröße
vec: Testmengenlänge

FIGUR 17